COMPUTER ARCHITECTURE

LIMKOKWING UNIVERSITY COLLEGE OF CREATIVE TECHNOLOGY

LECTURE 5

BY NDUMISO E. KHUMALO

INSTRUCTION SETS CHARACTERISTICS AND FUNCTIONS, ADDRESSING MODES AND FORMARTS

Objective

– Components of Instruction
– Types of Operands
– Intel x86
– Types of Instruction
– Addressing Modes
– Instruction formats
– Assembly Language

Components/Characteristics of Instructions

- The operation of the processor is determined by the instructions it executes, referred to as machine instructions or computer instructions.

- The collection of different instructions that the processor can execute is referred to as the processor’s ***instruction set***.

Components/Characteristics of Instructions

Elements of a Machine Instruction

- Each instruction must contain the information required by the processor for execution
- These elements are as follows:
  - 1. ***Operation code:*** Specifies the operation to be performed (e.g., ADD, I/O).
    - The operation is specified by a binary code, known as the operation code, or opcode

Components/Characteristics of Instructions

- These elements are as follows:
  - 2. ***Source operand reference:*** The operation may involve one or more source operands, that is, operands that are inputs for the operation
  - 3. ***Result operand reference:*** The operation may produce a result.
  - 4. ***Next instruction reference:*** This tells the processor where to fetch the next instruction after the execution of this instruction is complete.

Components/Characteristics of Instructions

- The address of the next instruction to be fetched could be either a real address or a virtual address, depending on the architecture.
  - Generally, the distinction is transparent to the instruction set architecture.
  - In most cases, the next instruction to be fetched immediately follows the current instruction.
    - So, there is no explicit reference to the next instruction
  - When an explicit reference is needed, the main or virtual memory address must be supplied

Components/Characteristics of Instructions

LIMKOKWING UNIVERSITY COLLEGE OF CREATIVE TECHNOLOGY

- Source and result operands can be in 1 of 4 areas:
  - ***1. Main or virtual memory:*** As with next instruction references, the main or virtual memory address must be supplied.
  - ***2. Processor register:*** With rare exceptions, a processor contains one or more registers that may be referenced by machine instructions.
    - If only one register exists, reference to it may be implicit.
    - If more than one register exists, then each register is assigned a unique name or number, and the instruction must contain the number of the desired register.

Components/Characteristics of Instructions

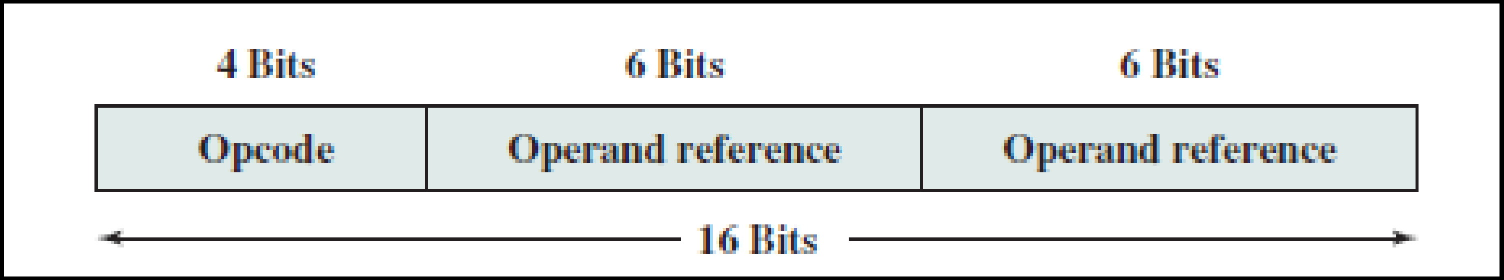
- Source and result operands can be in 1 of 4 areas:
  – ***3. Immediate:*** The value of the operand is contained in a field in the instruction being executed.

  – ***4. I/O device:*** The instruction must specify the I/O module and device for the operation.
    - If memory- mapped I/O is used, this is just another main or virtual memory address.

Components/Characteristics of Instructions

Instruction Representation

- Within the computer, each instruction is represented by a sequence of bits.
  - The instruction is divided into fields, corresponding to the constituent elements of the instruction
  - A simple instruction format



Components/Characteristics of Instructions

Instruction Representation

- With most instruction sets, more than one format is used.
  - During instruction execution, an instruction is read into an instruction register (IR) in the processor.
  - The processor must be able to extract the data from the various instruction fields to perform the required operation.

Components/Characteristics of Instructions

- Opcodes are represented by abbreviations, called mnemonics, that indicate the operation. Common examples include
  - ADD Add
  - SUB Subtract
  - MUL Multiply
  - DIV Divide
  - LOAD Load data from memory
  - STOR Store data to memory

Components/Characteristics of Instructions

- Operands are also represented symbolically. For example, the instruction
  - ADD R, Y may mean add the value contained in data location Y to the contents of register R
    - In this example, Y refers to the address of a location in memory, and R refers to a particular register.
    - Note that the operation is performed on the contents of a location, not on its address.
  - Thus, it is possible to write a machine-language program in symbolic form.

Components/Characteristics of Instructions

– Each symbolic opcode has a fixed binary representation, and the programmer specifies the location of each symbolic operand
  - For example, the programmer might begin with a list of definitions:

    X = 513

    Y = 514

    and so on.

– A simple program would accept this symbolic input, convert opcodes and operand references to binary form, and construct binary machine instructions.

Components/Characteristics of Instructions

- Machine- language programmers are rare to the point of nonexistence.
  - Most programs today are written in a high-level language or, failing that, assembly language
  - However, symbolic machine language remains a useful tool for describing machine instructions, and we will use it for that purpose.

Components/Characteristics of Instructions

Instruction Types

- Consider a high- level language instruction that could be expressed in a language such as BASIC or FORTRAN OR JAVA.
- For example,

  X = X + Y

  – This statement instructs the computer to add the value stored in Y to the value stored in X and put the result in X.

    - How might this be accomplished with machine instructions?

Components/Characteristics of Instructions

- Let us assume that the variables X and Y correspond to locations 513 and 514.
  - If we assume a simple set of machine instructions, this operation could be accomplished with three instructions:
    - 1. Load a register with the contents of memory location 513.
    - 2. Add the contents of memory location 514 to the register.
    - 3. Store the contents of the register in memory location 513.
  - As can be seen, the single BASIC instruction may require three machine instructions.

Components/Characteristics of Instructions

- This is typical of the relationship between a high- level language and a machine language.
  - A high-level language expresses operations in a concise algebraic form, using variables.
  - A machine language expresses operations in a basic form involving the movement of data to or from registers.

- A computer should have a set of instructions that allows the user to formulate any data processing task.
  - Another way to view it is to consider the capabilities of a high-level programming language.

Components/Characteristics of Instructions

- Any program written in a high- level language must be translated into machine language to be executed.
  - Thus, the set of machine instructions must be sufficient to express any of the instructions from a high-level language
- With this in mind we can categorize instruction types as follows:
  - ***Data processing:*** Arithmetic and logic instructions.
  - ***Data storage:*** Movement of data into or out of register and or memory locations.
  - ***Data movement:*** I/O instructions.
  - ***Control:*** Test and branch instructions.

Components/Characteristics of Instructions

- Arithmetic instructions provide computational capabilities for processing numeric data.
  - Logic (Boolean) instructions operate on the bits of a word as bits rather than as numbers; thus, they provide capabilities for processing any other type of data the user may wish to employ.
  - These operations are performed primarily on data in processor registers.
    - Therefore, there must be memory instructions for moving data between memory and the registers

Components/Characteristics of Instructions

LIMKOKWING UNIVERSITY COLLEGE OF CREATIVE TECHNOLOGY

- I/O instructions are needed to transfer programs and data into memory and the results of computations back out to the user.
- Test instructions are used to test the value of a data word or the status of a computation.
  - Branch instructions are then used to branch to a different set of instructions depending on the decision made.

Types of Operands

- Machine instructions operate on data. The most important general categories of data are
  - Addresses
  - Numbers
  - Characters
  - Logical data

Types of Operands

Numbers

- All machine languages include numeric data types
  - Even in nonnumeric data processing, there is a need for numbers to act as counters, field widths, and so forth.
  - An important distinction between numbers used in ordinary mathematics and numbers stored in a computer is that the latter are limited.
    - there is a limit to the magnitude of numbers representable on a machine
    - in the case of floating- point numbers, a limit to their precision.

Types of Operands

Numbers

- the programmer is faced with understanding the consequences of rounding, overflow, and underflow.
- Three types of numerical data are common in computers:
  - Binary integer or binary fixed point
  - Binary floating point
  - Decimal

Types of Operands

- Decimal Numbers
  – Although all internal computer operations are binary in nature, the human users of the system deal with decimal numbers.
    - Thus, there is a necessity to convert from decimal to binary on input and from binary to decimal on output
    - For applications in which there is a great deal of I/O and comparatively little, comparatively simple computation, it is preferable to store and operate on the numbers in decimal form.
  – The most common representation for this purpose is packed decimal or binary-coded decimal (BCD)

Types of Operands

Decimal Numbers

- With packed decimal, each decimal digit is represented by a 4-bit code, in the obvious way, with two digits stored per byte.
  - Thus, 0 = 0000, 1 = 0001, …….., 8 = 1000, and 9 = 1001
  - Note that this is a rather inefficient code because only 10 of 16 possible 4-bit values are used.
  - To form numbers, 4-bit codes are strung together, usually in multiples of 8 bits.
    - Thus, the code for 246 is 0000 0010 0100 0110.

Types of Operands

- This code is clearly less compact than a straight binary representation, but it avoids the conversion overhead.
- Negative numbers can be represented by including a 4-bit sign digit at either the left or right end of a string of packed decimal digits.
  - Standard sign values are 1100 for positive (+) and 1101 for negative (-).
    - Many machines provide arithmetic instructions for performing operations directly on packed decimal numbers

Types of Operands

Characters

- A common form of data is text or character strings
  - While textual data are most convenient for human beings, they cannot, in character form, be easily stored or transmitted by data processing and communications systems.
    - Such systems are designed for binary data
  - Thus, a number of codes have been devised by which characters are represented by a sequence of bits

Types of Operands

Characters

- the earliest common example of this is the Morse code
- Today, the most commonly used character code in the International Reference Alphabet (IRA), referred to in the United States as the American Standard Code for Information Interchange (ASCII)

Types of Operands

Characters

- Each character in this code is represented by a unique 7-bit pattern;
  - thus, 128 different characters can be represented.

- This is a larger number than is necessary to represent printable characters, and some of the patterns represent control characters

Types of Operands

Characters

- Each character in this code is represented by a unique 7-bit pattern;
  - thus, 128 different characters can be represented.
- This is a larger number than is necessary to represent printable characters, and some of the patterns represent control characters
  - Some of these control characters have to do with controlling the printing of characters on a page.
  - Others are concerned with communications procedures.

Types of Operands

Characters

- IRA-encoded characters are almost always stored and transmitted using 8 bits per character.
  – The eighth bit may be set to 0 or used as a parity bit for error detection.
    - In the latter case, the bit is set such that the total number of binary 1s in each octet is always odd (odd parity) or always even (even parity).

Types of Operands

Characters

- the IRA bit pattern 011XXXX, the digits 0 through 9 are represented by their binary equivalents, 0000 through 1001, in the rightmost 4 bits.
- This is the same code as packed decimal.
  - This facilitates conversion between 7-bit IRA and 4-bit packed decimal representation.
- Another code used to encode characters is the Extended Binary Code Decimal Interchange Code (EBCDIC)

Types of Operands

Characters

- EBCDIC is used on IBM mainframes.
  - It is an 8-bit code.
  - As with IRA, EBCDIC is compatible with packed decimal.
  - In the case of EBCDIC, the codes 11110000 through 11111001 represent the digits 0 through 9.

Types of Operands

Logical Data

- Normally, each word or other addressable unit (byte, halfword, and so on) is treated as a single unit of data
  - It is sometimes useful, however, to consider an n-bit unit as consisting of n 1-bit items of data, each item having the value 0 or 1.
  - When data are viewed this way, they are considered to be logical data.

Types of Operands

Logical Data

- There are two advantages to the bit-oriented view.
  - First, we may sometimes wish to store an array of Boolean or binary data items, in which each item can take on only the values 1 (true) and 0 (false).
    - With logical data, memory can be used most efficiently for this storage.
  - Second, there are occasions when we wish to manipulate the bits of a data item.
    - E.g., if floating-point operations are implemented in software, we need to be able to shift significant bits in some operations.

Types of Operands

Logical Data

- Another example: To convert from IRA to packed decimal, we need to extract the rightmost 4 bits of each byte.

– Note that, in the preceding examples, the same data are treated sometimes as logical and other times as numerical or text.

- The “type” of a unit of data is determined by the operation being performed on it.
- While this is not normally the case in high- level languages, it is almost always the case with machine language.

Intel x86

x86 Data Types

- The x86 can deal with data types of 8 (byte), 16 (word), 32 (doubleword), 64 (quadword), & 128 (double quadword) bits in length
- To allow maximum flexibility in data structures & efficient memory utilization, words need not be aligned at even-numbered addresses;
  - doublewords need not be aligned at addresses evenly divisible by 4;
  -

Intel x86

x86 Data Types

- quadwords need not be aligned at addresses evenly divisible by 8; and so on.
- However, when data are accessed across a 32-bit bus, data transfers take place in units of doublewords, beginning at addresses divisible by 4
- The processor converts the request for misaligned values into a sequence of requests for the bus transfer.
  - As with all of the Intel 80x86 machines, the x86 uses the little-endian style; that is, the least significant byte is stored in the lowest address

Intel x86

x86 Data Types

- The byte, word, doubleword, quadword, and double quadword are referred to as general data types.
  - In addition, the x86 supports an impressive array of specific data types that are recognized and operated on by particular instructions

- The next table summarizes these types.

Intel x86

x86 Data Types

Data Type	Description
General	Byte, word (16 bits), doubleword (32 bits), quadword (64 bits), and double quadword (128 bits) locations with arbitrary binary contents.
Integer	A signed binary value contained in a byte, word, or doubleword, using twos complement representation.
Ordinal	An unsigned integer contained in a byte, word, or doubleword.
Unpacked binary coded decimal (BCD)	A representation of a BCD digit in the range 0 through 9, with one digit in each byte.
Packed BCD	Packed byte representation of two BCD digits; value in the range 0 to 99.
Near pointer	A 16-bit, 32-bit, or 64-bit effective address that represents the offset within a segment. Used for all pointers in a nonsegmented memory and for references within a segment in a segmented memory.
Far pointer	A logical address consisting of a 16-bit segment selector and an offset of 16, 32, or 64 bits. Far pointers are used for memory references in a segmented memory model where the identity of a segment being accessed must be specified explicitly.
Bit field	A contiguous sequence of bits in which the position of each bit is considered as an independent unit. A bit string can begin at any bit position of any byte and can contain up to 32 bits.
Bit string	A contiguous sequence of bits, containing from zero to $2^{23} - 1$ bits.
Byte string	A contiguous sequence of bytes, words, or doublewords, containing from zero to $2^{23} - 1$ bytes.
Floating point	See Figure 12.4.
Packed SIMD (single instruction, multiple data)	Packed 64-bit and 128-bit data types.

Intel x86

x86 Operation Types

- The x86 provides a complex array of operation types, including a number of specialized instructions
  - The intent was to provide tools for the compiler writer to produce optimized machine language translation of high-level language programs.
  - Most of these are the conventional instructions found in most machine instruction sets, but several types of instructions are tailored to the x86 architecture and are of particular interest

Addressing Modes

- The address field or fields in a typical instruction format are relatively small.
- We would like to be able to reference a large range of locations in main memory or, for some systems, virtual memory
- To achieve this objective, a variety of addressing techniques has been employed.

Addressing Modes

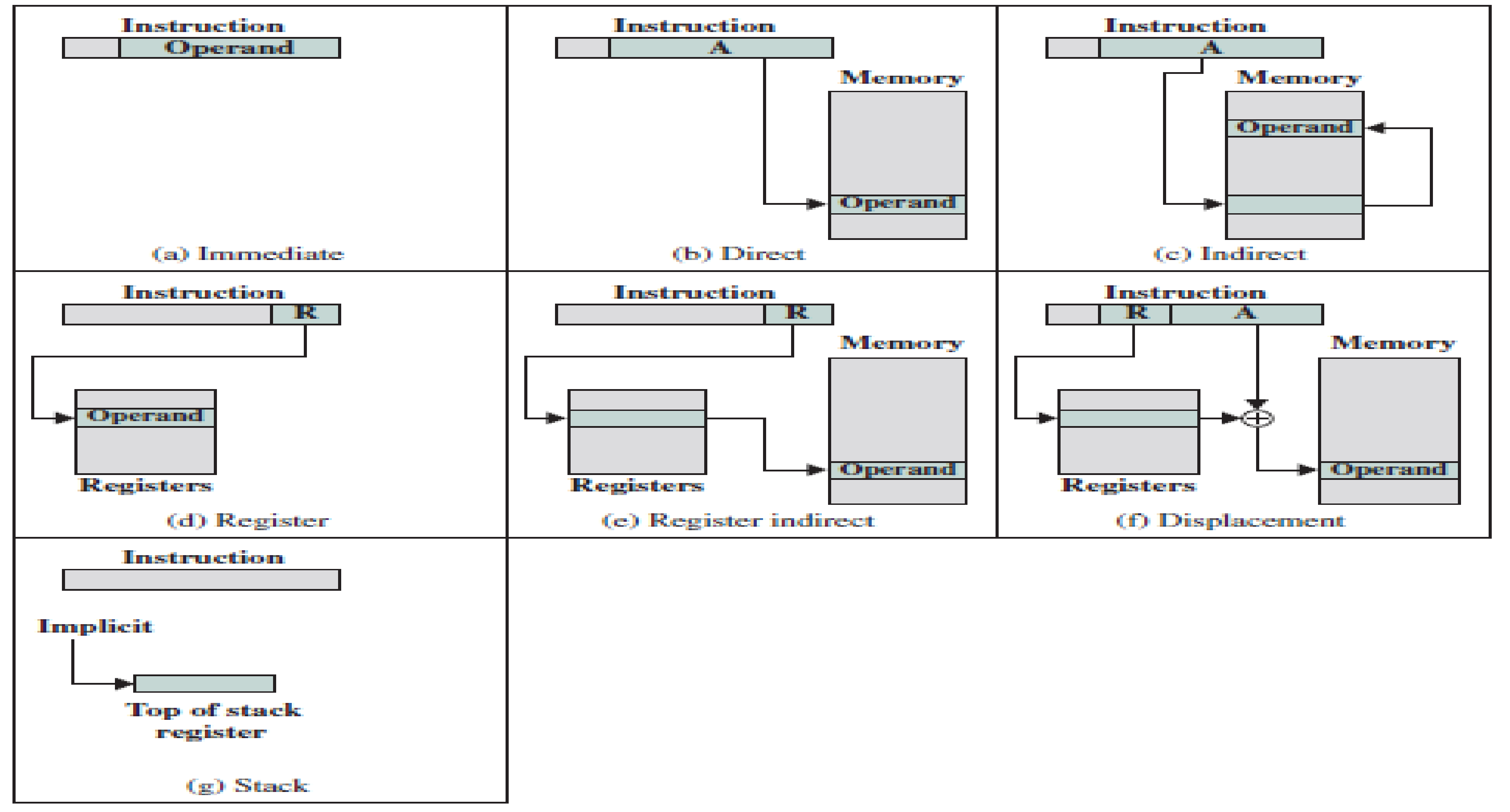
- These addressing techniques all involve :
  - some trade-off between address range and/or addressing flexibility, on one hand
  - the number of memory references in the instruction and/or the complexity of address calculation, on the other
  - the most common addressing techniques, or modes:
    - Immediate, Direct, Indirect, Register, Register indirect, Displacement, Stack

Addressing Modes

- Here, we use the following notation:
  - A = contents of an address field in the instruction
  - R = contents of an address field in the instruction that refers to a register
  - EA = actual (effective) address of the location containing the referenced operand
  - (X) = contents of memory location X or register X

Addressing Modes

- Addressing Modes



Addressing Modes

- Two comments need to be made.
  - First, virtually all computer architectures provide more than one of these addressing modes.
  - Second, in a system without virtual memory, the effective address will be either a main memory address or a register.
    - In a virtual memory system, the effective address is a virtual address or a register.
    - The actual mapping to a physical address is a function of the memory management unit (MMU) and is invisible to the programmer.

Addressing Modes

- Basic Addressing Modes

Mode	Algorithm	Principal Advantage	Principal Disadvantage
Immediate	Operand = A	No memory reference	Limited operand magnitude
Direct	EA = A	Simple	Limited address space
Indirect	EA = (A)	Large address space	Multiple memory references
Register	EA = R	No memory reference	Limited address space
Register indirect	EA = (R)	Large address space	Extra memory reference
Displacement	EA = A + (R)	Flexibility	Complexity
Stack	EA = top of stack	No memory reference	Limited applicability

Addressing Modes

Immediate Addressing

- The simplest form of addressing is immediate addressing, in which the operand value is present in the instruction

  Operand = A

- This mode can be used to define and use constants or set initial values of variables.
  - Typically, the number will be stored in twos complement form; the leftmost bit of the operand field is used as a sign bit.

Addressing Modes

Immediate Addressing

- When the operand is loaded into a data register, the sign bit is extended to the left to the full data word size

- In some cases, the immediate binary value is interpreted as an unsigned nonnegative integer

Addressing Modes

Immediate Addressing

- The advantage of immediate addressing is that no memory reference other than the instruction fetch is required to obtain the operand,
  - thus saving 1 memory or cache cycle in the instruction cycle
- The disadvantage is that the size of the number is restricted to the size of the address field, which, in most instruction sets, is small compared with the word length.

Addressing Modes

Direct Addressing

- A very simple form of addressing is direct addressing, in which the address field contain the effective address of the operand:

$$EA = A$$

- The technique was common in earlier generations of computers but is not common on contemporary architectures.

Addressing Modes

Direct Addressing

- It requires only one memory reference and no special calculation.

- The obvious limitation is that it provides only a limited address space.

Addressing Modes

Indirect Addressing

- With direct addressing, the length of the address field is usually less than the word length, thus limiting the address range.
- One solution is to have the address field refer to the address of a word in memory, which in turn contains a full-length address of the operand.
- This is known as indirect addressing:

$$EA = (A)$$

Addressing Modes

Indirect Addressing

- As defined earlier, the parentheses are to be interpreted as meaning contents of.
- The obvious advantage of this approach is that for a word length of N, an address space of 2^N is now available.
  - The disadvantage is that instruction execution requires two memory references to fetch the operand: one to get its address and a second to get its value.

Addressing Modes

Register Addressing

- Register addressing is similar to direct addressing.
- The only difference is that the address field refers to a register rather than a main memory address:

$$EA = R$$

- To clarify, if the contents of a register address field in an instruction is 5, then register R5 is the intended address, and the operand value is contained in R5

Addressing Modes

Register Addressing

- The advantages of register addressing are that :
  - (1) only a small address field needed in the instruction,
  - (2) no time-consuming memory references are required.

- The disadvantage of register addressing is that the address space is very limited.

Addressing Modes

Register Indirect Addressing

- Just as register addressing is analogous to direct addressing, register indirect addressing is analogous to indirect addressing.
- In both cases, the only difference is whether the address field refers to a memory location or a register.
- Thus, for register indirect address:

$$EA = (R)$$

Addressing Modes

Register Indirect Addressing

- The advantages and limitations of register indirect addressing are basically the same as for indirect addressing.
- In both cases, the address space limitation (limited range of addresses) of the address field is overcome by having that field refer to a word-length location containing an address.
  - In addition, register indirect addressing uses one less memory reference than indirect addressing

Addressing Modes

Displacement Addressing

- A very powerful mode of addressing combines the capabilities of direct addressing and register indirect addressing.
- It is known by a variety of names depending on the context of its use, but the basic mechanism is the same.
- We will refer to this as displacement addressing:

$$EA = A + (R)$$

Addressing Modes

Displacement Addressing

- Displacement addressing requires that the instruction have two address fields, at least one of which is explicit.
- The value contained in one address field (value = A) is used directly.
  - The other address field, or an implicit reference based on opcode, refers to a register whose contents are added to A to produce the effective address.

Addressing Modes

Displacement Addressing

- three of the most common uses of displacement addressing:
  - Relative addressing
  - Base-register addressing
  - Indexing

- READ ON THESE

Addressing Modes

Stack Addressing

- a stack is a linear array of locations.
  - It is sometimes referred to as a pushdown list or last-in-first-out queue.
- The stack is a reserved block of locations.
- Items are appended to the top of the stack so that, at any given time, the block is partially filled.
- Associated with the stack is a pointer whose value is the address of the top of the stack.

Addressing Modes

Stack Addressing

- Alternatively, the top two elements of the stack may be in processor registers, in which case the stack pointer references the third element of the stack
- The stack pointer is maintained in a register.
  - Thus, references to stack locations in memory are in fact register indirect addresses
  - stack mode of addressing is a form of implied addressing.
    - The machine instructions need not include a memory reference but implicitly operate on the top of the stack.

Instruction Formats

- An instruction format defines the layout of the bits of an instruction, in terms of its constituent fields
- An instruction format must include an opcode and, implicitly or explicitly, zero or more operands
  - Each explicit operand is referenced using one of the addressing modes
- The format must, implicitly or explicitly, indicate the addressing mode for each operand.
  - For most instruction sets, more than one instruction format is used.

Instruction Formats : key design issues

Instruction Length

- The most basic design issue to be faced is the instruction format length.
  - This decision affects, and is affected by, memory size, memory organization, bus structure, processor complexity, and processor speed
  - This decision determines the richness and flexibility of the machine as seen by the assembly-language programmer

Instruction Formats : key design issues

Instruction Length

- opcodes, operands, addressing modes, address range etc. require bits and push in the direction of longer instruction lengths.
  - But longer instruction length may be wasteful.
  - A 64-bit instruction occupies twice the space of a 32-bit instruction but is probably less than twice as useful.

Instruction Formats : key design issues

Allocation of Bits

- An equally difficult issue is how to allocate the bits in that format.
- The trade-offs here are complex.
  - For a given instruction length, there is clearly a trade-off between the number of opcodes and the power of the addressing capability.
    - More opcodes obviously mean more bits in the opcode field
    - For an instruction format of a given length, this reduces the number of bits available for addressing

Instruction Formats : key design issues

Variable-Length Instructions

- The examples we have looked at so far have used a single fixed instruction length, and we have implicitly discussed trade-offs in that context.
- But the designer may choose instead to provide a variety of instruction formats of different lengths
  - This tactic makes it easy to provide a large repertoire of opcodes, with different opcode lengths

Instruction Formats : key design issues

Variable-Length Instructions

- Addressing can be more flexible, with various combinations of register and memory references plus addressing modes.
  - With variable- length instructions, these many variations can be provided efficiently and compactly
- The principal price to pay for variable-length instructions is an increase in the complexity of the processor

Assembly Language

- A processor can understand and execute machine instructions.
- Such instructions are simply binary numbers stored in the computer.

- If a programmer wished to program directly in machine language, then it would be necessary to enter the program as binary data.

Assembly Language

- Consider the simple BASIC statement

  N = I + J + K

- Suppose we wished to program this statement in machine language and to initialize I, J, and K to 2, 3, and 4, respectively

Assembly Language

- Binary program

Address	Contents			
101	0010	0010	101	2201
102	0001	0010	102	1202
103	0001	0010	103	1203
104	0011	0010	104	3204
201	0000	0000	201	0002
202	0000	0000	202	0003
203	0000	0000	203	0004
204	0000	0000	204	0000

Assembly Language

- The program starts in location 101 (hexadecimal).
  - Memory is reserved for the four variables starting at location 201.
  - The program consists of four instructions
    - 1. Load the contents of location 201 into the AC.
    - 2. Add the contents of location 202 to the AC.
    - 3. Add the contents of location 203 to the AC.
    - 4. Store the contents of the AC in location 204.

Assembly Language

- This is clearly a tedious & very error-prone process.
- A slight improvement is to write the program in hexadecimal rather than binary notation
- Hexadecimal program

Address	Contents
101	2201
102	1202
103	1203
104	3204
201	0002
202	0003
203	0004
204	0000

Assembly Language

- We could write the hexadecimal program as a series of lines.
  - Each line contains the address of a memory location and the hexadecimal code of the binary value to be stored in that location
  - Then we need a program that will accept this
  - input, translate each line into a binary number, and store it in the specified location

Assembly Language

- For more improvement, we can make use of the symbolic name or mnemonic of each instruction.
  - This results in the symbolic program
  - Symbolic program

Address	Instruction	
101	LDA	201
102	ADD	202
103	ADD	203
104	STA	204
201	DAT	2
202	DAT	3
203	DAT	4
204	DAT	0

Assembly Language

- For the symbolic program, each line of input still represents one memory location.
  - Each line consists of three fields, separated by spaces
  - The first field contains the address of a location.
  - For an instruction, the second field contains the three-letter symbol for the opcode.
  - If it is a memory-referencing instruction, then a third field contains the address.

Assembly Language

- To store arbitrary data in a location, we invent a pseudoinstruction with the symbol DAT.
  - This is merely an indication that the third field on the line contains a hexadecimal number to be stored in the location specified in the first field.
- For this type of input we need a slightly more complex program.
- The program accepts each line of input, generates a binary number based on the second and third (if present) fields, and stores it in the location specified by the first field.

Assembly Language

- The use of a symbolic program makes life much easier but is still awkward
  - In particular, we must give an absolute address for each word.
  - This means that the program and data can be loaded into only one place in memory, and we must know that place ahead of time
  - Worse, suppose we wish to change the program some day by adding or deleting a line.
    - This will change the addresses of all subsequent words.

Assembly Language

- A much better system, and one commonly used, is to use symbolic addresses.
- Assembly Language using symbolic addresses

Label	Operation	Operand
FORMUL	LDA	I
	ADD	J
	ADD	K
	STA	N
I	DATA	2
J	DATA	3
K	DATA	4
N	DATA	0

Assembly Language

- In this system, each line still consists of three fields.
  - The first field is still for the address, but a symbol is used instead of an absolute numerical address.
  - Some lines have no address, implying that the address of that line is one more than the address of the previous line.
  - For memory-reference instructions, the third field also contains a symbolic address

Assembly Language

- With this last refinement, we have an assembly language.
  - Programs written in assembly language (assembly programs) are translated into machine language by an assembler.
  - This program must not only do the symbolic translation discussed earlier but also assign some form of memory addresses to symbolic addresses.

Assembly Language

- The development of assembly language was a major milestone in the evolution of computer technology.
  - It was the first step to the high-level languages in use today
  - Although few programmers use assembly language, virtually all machines provide one.
  - They are used, if at all, for systems programs such as compilers and I/O routines.
  - Appendix B in the textbook provides a more detailed examination of assembly language

THE END