COMPUTER ARCHITECTURE

LIMKOKWING UNIVERSITY COLLEGE OF CREATIVE TECHNOLOGY

LECTURE 8

BY NDUMISO E. KHUMALO

ASSEMBLY LANGUAGE, RISC, SISC AND PARALLEL PROCESSING

Objective

– Assembly Language
– RISC and CISC
– Parallel Processing

Assembly Language

- Assembly language is a low-level programming language for a computer or other programmable device specific to a particular computer architecture in contrast to most high-level programming languages, which are generally portable across multiple systems.
- Assembly language is converted into executable machine code by a utility program referred to as an assembler like NASM, MASM, etc.

Assembly Language

- Each personal computer has a microprocessor that manages the computer's arithmetical, logical, and control activities.
- Each family of processors has its own set of instructions for handling various operations such as getting input from keyboard, displaying information on screen and performing various other jobs.
- These set of instructions are called 'machine language instructions'.

Assembly Language

- A processor understands only machine language instructions, which are strings of 1's and 0’s.
- However, machine language is too obscure and complex for using in software development.

- So, the low-level assembly language is designed for a specific family of processors that represents various instructions in symbolic code and a more understandable form.

Assembly Language

- Having an understanding of assembly language makes one aware of:
  - How programs interface with OS, processor, and BIOS;
  - How data is represented in memory and other external devices;
  - How the processor accesses and executes instruction;
  - How instructions access and process data;
  - How a program accesses external devices.

Assembly Language

- Other advantages of using assembly language are:
  - It requires less memory and execution time;
  - It allows hardware-specific complex jobs in an easier way;
  - It is suitable for time-critical jobs;
  - It is most suitable for writing interrupt service routines and other memory resident programs.

Assembly Language

- An assembly program can be divided into three sections:
  - 1. The data section
    - The data section is used for declaring initialized data or constants.
    - This data does not change at runtime.
    - You can declare various constant values, file names, or buffer size, etc., in this section.
    - The syntax for declaring data section is:

      section.data

Assembly Language

– 2. The bss Section
  - The bss section is used for declaring variables. The syntax for declaring bss section is:

```
section.bss
```

– 3. The text section
  - The text section is used for keeping the actual code.
  - This section must begin with the declaration global _start, which tells the kernel where the program execution begins.
  - The syntax for declaring text section is:

```
section.text
        global _start
_start:
```

Assembly Language

- Comments
  - Assembly language comment begins with a semicolon (;).
  - It may contain any printable character including blank. It can appear on a line by itself, like:

```
; This program displays a message on screen
```

  - or, on the same line along with an instruction, like –

```
add eax, ebx     ; adds ebx to eax
```

Assembly Language

- Assembly language programs consist of three types of statements:
  - 1. Executable instructions or instructions
    - They tell the processor what to do.
    - Each instruction consists of an operation code (opcode).
    - Each executable instruction generates one machine language instruction.
  - 2. Macros.
    - Macros are basically a text substitution mechanism.

Assembly Language

- Assembly language programs consist of three types of statements:
  - 3. The assembler directives or pseudo-ops
    - tell the assembler about the various aspects of the assembly process.
    - These are non-executable and do not generate machine language instructions.

Assembly Language

- Assembly language statements are entered one statement per line.
- Each statement follows the following format:

  [label]   mnemonic   [operands]   [;comment]

  – The fields in the square brackets are optional.

  – A basic instruction has two parts:
    - the first one is the name of the instruction (or the mnemonic), which is to be executed
    - the second are the operands or the parameters of the command.

Assembly Language

- The following are some examples of typical assembly language statements

```
INC COUNT        ; Increment the memory variable COUNT

MOV TOTAL, 48    ; Transfer the value 48 in the
                 ; memory variable TOTAL

ADD AH, BH       ; Add the content of the
                 ; BH register into the AH register

AND MASK1, 128   ; Perform AND operation on the
                 ; variable MASK1 and 128

ADD MARKS, 10    ; Add 10 to the variable MARKS
MOV AL, 10       ; Transfer the value 10 to the AL register
```

Assembly Language

- The following assembly language code displays the string 'Hello World' on the screen

```
section .text
   global _start     ;must be declared for linker (ld)

_start:             ;tells linker entry point
   mov  edx,len     ;message length
   mov  ecx,msg     ;message to write
   mov  ebx,1       ;file descriptor (stdout)
   mov  eax,4       ;system call number (sys_write)
   int  0x80        ;call kernel

   mov  eax,1       ;system call number (sys_exit)
   int  0x80        ;call kernel

section .data
msg db 'Hello, world!', 0xa  ;string to be printed
len equ $ - msg     ;length of the string
```

Assembly Language

- The registers are defined as:
  - EAX - Accumulator Register
  - EBX - Base Register
  - ECX - Counter Register
  - EDX - Data Register
  - ESI - Source Index
  - EDI - Destination Index
  - EBP - Base Pointer
  - ESP - Stack Pointer

Assembly Language

- To set up your environment for assembly language on Windows:
  - 1. Open command prompt (cmd) as an Administrator
  - 2. Type *wsl --install* and press Enter
  - 3. Enter your new Linux username and password when prompted to do so and press Enter
  - 4. Restart your machine
  - 5. Open command prompt (cmd) as an administrator and type *wsl* and press Enter

Assembly Language

- To set up your environment for assembly language on Windows:
  - 6. Type *cd /mnt/c* then press Enter to go to drive C:
  - 7. Create a folder named nasm by typing *mkdir nasm* and press Enter
  - 8. Type *cd nasm*

Assembly Language

– 9. To install nasm, follow these steps:
  - i. First check if it is already installed, by typing *whereis nasm* and press Enter. It should show the path where nasm is installed. If not continue to ii below.
  - ii. Type *sudo apt-get update* and press Enter. Supply your password if prompted to do so. Wait until it's done.
  - iii. Type *sudo apt-get -y install nasm* and press Enter. Supply your password if prompted to do so. Wait until it's done

– 10. Type *whereis nasm* and press enter to check if it has been installed
  - It should show the path where nasm is installed

Assembly Language

- Compiling and Linking an Assembly Program in NASM
  - Make sure you have set the path of nasm and ld binaries in your PATH environment variable. Now, take the following steps for compiling and linking the above program –
    1. Type the code using a text editor and save it as *hello.asm*
    2. Make sure that you are in the same directory as where you saved hello.asm
       1. Preferably, save it in the nasm folder created earlier

Assembly Language

- 4. To assemble the program, type

  nasm -f elf hello.asm

- 5. If there is any error, you will be prompted about that at this stage. Otherwise, an object file of your program named *hello.o* will be created.

- 6. To link the object file and create an executable file named hello, type *ld -m elf_i386 -s -o hello hello.o*

- 7. Execute the program by typing *./hello*

Assembly Language

- Download the functions.asm file from https://classroom.google.com/c/NTkwNjIxNzg4ODc5/m/NjQ0OTgxNzI5NDI4/details to help you with the next two problems

- Copy it to the nasm folder

Assembly Language; Addition

- Type the following program. Assembly it and run it.

```
%include 'functions.asm'
SECTION .text
global  _start
_start:
    mov     eax, 30     ; move our first number into eax
    mov     ebx, 9      ; move our second number into ebx
    add     eax, ebx    ; add ebx to eax
    call    iprintLF    ; call our integer print with linefeed function
    call    quit
```

Assembly Language: Subtraction

- Type the following program. Assembly it and run it.

```
%include 'functions.asm'
SECTION .text
global  _start
_start:
    mov     eax, 30     ; move our first number into eax
    mov     ebx, 9      ; move our second number into ebx
    sub     eax, ebx    ; subtract ebx from eax
    call    iprintLF    ; call our integer print with linefeed function
    call    quit
```

Assembly Language: Multiplication

- Type the following program. Assembly it and run it.

```
%include 'functions.asm'
SECTION .text
global  _start
_start:
    mov     eax, 30    ; move our first number into eax
    mov     ebx, 9     ; move our second number into ebx
    mul     ebx   ; multiply ebx by eax
    call    iprintLF   ; call our integer print with linefeed function
    call    quit
```

Assembly Language: User input

- Type the following program. Assembly it and run it.

```
%include        'functions.asm'

SECTION .data
msg1        db      'Please enter your name: ', 0h      ; message string asking user for input
msg2        db      'Hello, ', 0h                       ; message string to use after user has entered their name

SECTION .bss
sinput:     resb    255                                 ; reserve a 255 byte space in memory for the users input string

SECTION .text
global  _start

_start:

    mov     eax, msg1
    call    sprint

    mov     edx, 255        ; number of bytes to read
    mov     ecx, sinput     ; reserved space to store our input (known as a buffer)
    mov     ebx, 0          ; read from the STDIN file
    mov     eax, 3          ; invoke SYS_READ (kernel opcode 3)
    int     80h

    mov     eax, msg2
    call    sprint

    mov     eax, sinput     ; move our buffer into eax (Note: input contains a linefeed)
    call    sprint          ; call our print function

    call    quit
```

RISC and CISC

- What are RISCs and why do we need them?
  - RISC architectures represent an important innovation in the area of computer organization.
  - The RISC architecture is an attempt to produce more CPU power by simplifying the instruction set of the CPU.

- The opposed trend to RISC is that of complex instruction set computers (CISC).

RISC and CISC

- Both RISC and CISC architectures have been developed as an attempt to cover the semantic gap
- The Main Characteristics of RISC Architectures
  - The instruction set is limited and includes only simple instructions.
  - The goal is to create an instruction set containing instructions that execute quickly; most of the RISC instructions are executed in a single machine cycle (after fetched and decoded).

RISC and CISC

– Pipeline operation (without memory reference):
– RISC instructions, being simple, are hard-wired, while CISC architectures have to use microprogramming in order to implement complex instructions.
– Having only simple instructions results in reduced complexity of the control unit and the data path; as a consequence, the processor can work at a high clock frequency.

RISC and CISC

- The instruction set is limited and includes only simple instructions.
- The goal is to create an instruction set containing instructions that execute quickly; most of the RISC instructions are executed in a single machine cycle (after fetched and decoded).
- The pipelines are used efficiently if instructions are simple and of similar execution time.

RISC and CISC

– Complex operations on RISCs are executed as a sequence of simple RISC instructions. In the case of CISCs they are executed as one single or a few complex instruction.

RISC and CISC

- Are RISCs Really Better than CISCs?
  - RISC architectures have several advantages
  - However, a definitive answer to the above question is difficult to give.
  - A lot of performance comparisons have shown that benchmark programs are really running faster on RISC processors than on processors with CISC characteristics.

RISC and CISC

- Are RISCs Really Better than CISCs?
  - However, it is difficult to identify which feature of a processor produces the higher performance. Some "CISC fans" argue that the higher speed is not produced by the typical RISC features but because of technology, better compilers, etc.
  - An argument in favor of the CISC: the simpler instruction set of RISC processors results in a larger memory requirement compared to the similar program compiled for a CISC architecture

RISC and CISC

- Are RISCs Really Better than CISCs?
  - Most of the current processors are not typical RISCs or CISCs but try to combine advantages of both approaches

Parallel Processing

Why Parallel Computation?

- The need for high performance!

- Two main factors contribute to high performance of modern processors:
  - 1. Fast circuit technology
  - 2. Architectural features:
    - large caches, multiple fast buses, pipelining, superscalar architectures (multiple funct. units)

Parallel Processing

- However: Computers running with a single CPU, often are not able to meet performance needs in certain areas:
  - 1. Fluid flow analysis and aerodynamics;
  - 2. Simulation of large complex systems, for example in physics, economy, biology, technic;
  - 3. Computer aided design
  - 4. Multimedia.

Parallel Processing

- Applications in those domains are characterized by a very high amount of numerical computation and/or a high quantity of input data.

Parallel Processing

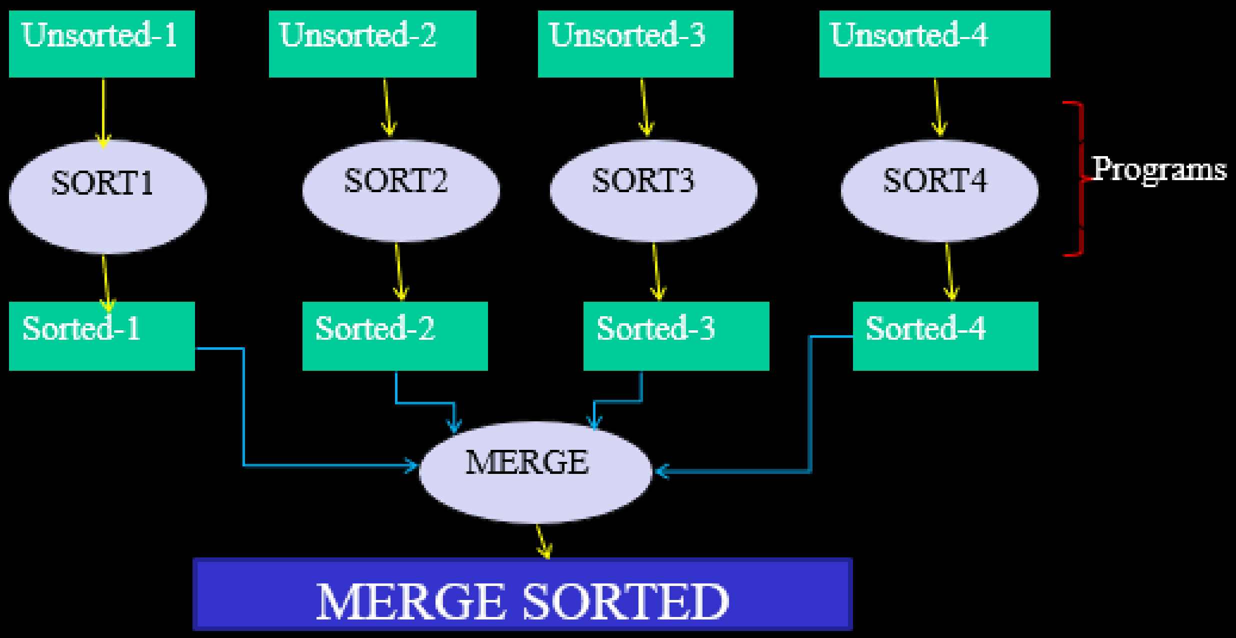
- A Solution: Parallel Computers
  - One solution to the need for high performance: architectures in which several CPUs are running in order to solve a certain application.
  - Such computers have been organized in very different ways.

Parallel Processing

- Some key features:
  - number and complexity of individual CPUs
  - availability of common (shared memory)
  - interconnection topology
  - performance of interconnection network
  - I/O devices

Parallel Processing

- Parallel Programs : Parallel sorting



Parallel Processing

- HIGH PERFOMANCE COMPUTING
  - There are different ways to classify parallel computers. One of the more widely used classifications, in use since 1966, is called Flynn's Taxonomy.
  - Flynn's taxonomy distinguishes multi-processor computer architectures according to how they can be classified along the two independent dimensions of Instruction and Data.
  - Each of these dimensions can have only one of two possible states: Single or Multiple.

Parallel Processing

- HIGH PERFOMANCE COMPUTING
  – The matrix below defines the 4 possible classifications according to Flynn

SISD	**SIMD**
MISD	MIMD

  – Flynn’s Classification of Computer Architectures is based on the nature of the instruction flow executed by the computer and that of the data flow on which the instructions operate.

Parallel Processing

- HIGH PERFOMANCE COMPUTING
  - 1. Single Instruction stream, Single Data stream (SISD)
  - 2. Single Instruction stream, Multiple Data stream (SIMD)

Parallel Processing

- SIMD (Single-Instruction Multi-Data)
  - All processors in a parallel computer execute the same instructions but operate on different data at the same time.
  - Only one program can be run at a time.
  - Processors run in synchronous, lockstep function
  - Shared or distributed memory
  - Less flexible in expressing parallel algorithms, usually exploiting parallelism on array operations, e.g. F90

Parallel Processing

- MIMD (Multi-Instruction Multi-data)
  - All processors in a parallel computer can execute different instructions and operate on different data at the same time.
  - Parallelism achieved by connecting multiple processors together
  - Shared or distributed memory
  - Different programs can be run simultaneously
  - Each processor can perform any operation regardless of what other processors are doing.

THE END

-