**CSC2231 COMPUTER ARCHITECTURE**

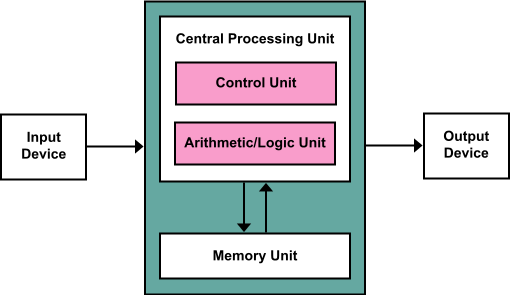
**Performance Issues**

Over the Years, The cost of computing systems continues to drop dramatically, while the performance and capacity of those systems continue to rise equally dramatically. Today’s laptops have the computing power of an IBM mainframe from 10 or 15 years ago. Thus, we have virtually “free” computer power. Processors are so inexpensive that we now have microprocessors we throw away.

**Microprocessor Speed:** What gives Intel x86 processors or IBM mainframe computers such mind-boggling power is the relentless pursuit of speed by processor chip manufacturers? The evolution of these machines continues to bear out.

The quest for better performance gives rise to the following techniques use by the processor.

**Processor**



**Performance**

The execution of an instruction involves multiple stages these includes.

1. Fetch the instruction
2. Detect the instruction
3. Execute the instruction

**Pipelining:** This enables a processor to work simultaneously on multiple instruction at same time by performing a different stage of each instruction at same time.

The processor overlaps operation by moving data's of instruction into a conceptual pipe with all stages been executed simultaneously.

* When an instruction is on decode another is on execute.

**Branch Prediction:** This process looks ahead in the instruction code fetch from memory and predicts which instruction or group of instructions are likely to be executed next. If the processors guess right most of the time, it can pre-fetch the correct instruction and buffer them so that the processor is kept busy.

* Super Scale Execution: This is the ability to issue more than one instruction in every processor clock cycle. In effect multiple parallel pipeline are used.
* Data Flow Analysis: The processor analyses which instruction are dependent on each other’s result or data to create an optimal schedule of instruction
* Speculative Execution: combining both branch prediction and data flow analysis, some processor execute instruction ahead or their actually appearance in the program execution.

These and many more helps the processor very fast and keeps the process busy.

**Performance Balance**

* While processor power has raced ahead at breakneck speed, other critical components of the computer have not kept up. The result is a need to look for performance balance: an adjustment/tuning of the organization and architecture to compensate for the mismatch among the capabilities of the various components.
* If memory or the pathway fails to keep pace with the processor’s insistent demands, the processor stalls in a wait state, and valuable processing time is lost.
* A system architect can attack this problem in a number of ways, all of which are reflected in contemporary computer designs.
* Increase the interconnect bandwidth between processors and memory by using higher-speed buses and a hierarchy of buses to buffer and structure data flow.
* Reduce the frequency of memory access by incorporating increasingly complex and efficient cache structures between the processor and main memory. This includes the incorporation of one or more caches on the processor chip as well as on an off-chip cache close to the processor chip.
* Change the DRAM interface to make it more efficient by including a cache1 or other buffering scheme on the DRAM chip.

**Execution Cycle**

* In evaluating processor hardware and setting requirements for new systems, performance is one of the key parameters to consider, along with cost, size, security, reliability, and, in some cases, power consumption.
* In measuring processor and performance speed we look at some traditional measures of processor speed.
* **Clock Speed/Clock Circle:** Operations performed by a processor, such as fetching an instruction, decoding the instruction, performing an arithmetic operation, and so on, are governed by a system clock. Typically, all operations begin with the pulse of the clock. Thus, at the most fundamental level, the speed of a processor is dictated by the pulse frequency produced by the clock, measured in cycles per second, or Hertz (Hz).
* Everything computer does must be completed before the start of the next cycle (altho program execution could be pipelined – split up) – A machine running at 5GHz receives 5 x 109 clock signals per second (one pulse lasts 0.0000000005 seconds)
* The execution of an instruction involves a number of discrete steps, such as fetching the instruction from memory, decoding the various portions of the instruction, loading and storing data, and performing arithmetic and logical operations. Thus, most instructions on most processors require multiple clock cycles to complete. Some instructions may take only a few cycles, while others require dozens. In addition, when pipelining is used, multiple instructions are being executed simultaneously.

Time

Usually called the Fetch-Decode-Execute cycle

**Instruction Cycle**

* We are all familiar with the clock cycle
* Instruction execution cycle is triggered by the clock cycle
  + Every stage of the instruction cycle is triggered by successive clock pulses
  + Exact timing depends on the details of a machine type
  + Complete instruction cycle usually takes several clock cycles to execute

Instruction cycle is divided into several stages

* In some machines, some stages are executed simultaneously
* The stages are common to most architectures

The steps that are followed when an instruction is executed are:

1. Inspect the program counter (PC) to find the address of the next instruction
2. Load the next instruction from the memory into the instruction register (IR)
3. Update the PC to point the next instruction to be fetched
4. Determine the type of instruction fetched
5. If the instruction requires data from the memory, determine its address
6. Fetch the data from memory into one of the CPU registers
7. Execute the instruction
8. Return to step a. for the next instruction

**Starting a Program**

1. When a piece of code is run, the first thing that must be done is loading the code into memory
2. Once in memory, it can be executed
3. Instructions and data associated with program will each occupy a block of memory.
4. Exact location is determined by operating system
5. Memory address of the 1st instruction is called the entry- point
6. Running a new piece of code commences by loading the entry-point into PC

PC = entry point;

**Executing an Instruction**

* The entry-point is only the address of the first instruction (we, however need the instruction itself)
* At the start of the next clock cycle, the CPU issues a request to the memory (thru the load/store unit)
  + Load/store unit sends the memory address and a request to read from

**The memory thru the address bus**

* Later, the instruction will be received from the memory by the load/store unit
  + The instruction is put into the Instruction Register (IR)

**IR = memory (PC);**

How long this takes depends on how fast the memory is

* Could be just in time for next clock cycle
* Could be many cycles later (if the instruction is in slow memory)
* At this point, the value of the PC is changed to point to the next instruction
  + Usually, this is just an increment
* But if there is a jump/branch in the code, the increment will be different
* The CPU then acts on the instruction in the IR
* 1st , the type of instruction is determined by the control unit
* The instruction could be one which
  + Move data around (e.g. a=b;)
  + Combine to operands (e.g. c=a+b;)
  + Manipulate one operand (e.g. bit shift/rotation)
  + Does a test/comparison and changes the program flow
* Other type of instructions are used to
  + Call in a new procedure or function (need to change PC)
  + Do I/O
  + Control loops
  + Do other special operation
* Once the type of instruction has been deduced, any data needed must be fetched from the memory
  + To do a + b, the add instruction needs to fetch data items a and b from the memory
  + This is done in the same way as the instruction was fetched
  + The operating system has assigned each variable a memory address
* Each variable is fetched from the memory and loaded into one of the local registers within the CPU
  + Some instructions on some machines can operate directly on the contents of the memory
* Once the variable is loaded, the control unit configures the ALU (or any appropriate functional unit) to execute the operation
* The result of the operation is usually stored in one of the registers but can be written directly to the memory in some architectures
* Some operations change the PC differently

**Examples of instruction set**

**ADD** - Add two numbers together.

**COMPARE** - Compare numbers.

**IN** - Input information from a device, e.g., keyboard.

**JUMP** - Jump to designated RAM address.

**JUMP IF** - Conditional statement that jumps to a designated RAM address.

**LOAD** - Load information from RAM to the CPU.

**OUT** - Output information to device, e.g., monitor.

**STORE** - Store information to RAM.

Once the type of instruction has been deduced, any data needed must be fetched from the memory

* To do a + b, the add instruction needs to fetch data items a and b from the memory
* This is done in the same way as the instruction was fetched
* The operating system has assigned each variable a memory address

Each variable is fetched from the memory and loaded into one of the local registers within the CPU

Some instructions on some machines can operate directly on the contents of the memory

* Once the variable is loaded, the control unit configures the ALU (or any appropriate functional unit) to execute the operation
* The result of the operation is usually stored in one of the registers but can be written directly to the memory in some architectures
* Some operations change the PC differently, e.g Branches and Jumps

**Example**

if (a==b)

...run the code starting at address x (PC=x)

else

...run the code starting at address y (PC=y) end

The result of the conditional determines which branch the program takes

If we reach the last statement of the loop, we need to jump back to the start

This also requires that the PC be changed

**Next Instruction**

Once the result of the operation is known, and any changes to the PC is made, the next instruction can be executed

* It follows the same procedure, exactly
* Instruction cycle is simple, we can write a simple pseudo- code to describe the cycle smore aptly

**Pseudo-code for Instruction Cycle**

PC = entry\_point; //set the starting point While (program is\_running) {

//load next instruction

//increment PC

IR = memory(PC); PC = PC + 1;

itype = type(IR); //find type of instruction

data\_loc = find\_data(IR,itype);

//get data location, (-1 if none)

if (data\_loc >= 0) {//if data is needed data = memory(data\_loc) //fetch it

}

execute(itype,data) //execute the instruction

}

**Instruction Set/ Instruction Set Architecture**

The instruction set, also called ISA (Instruction Set Architecture), an instruction set is a group of commands for a CPU in machine language.

The possible instructions for a CPU or a subset of instructions to enhance its performance in certain situations. The instruction set provides commands to the processor, to tell it what it needs to do. The instruction set consists of addressing modes, instructions, native data types, registers, memory architecture, interrupt, and exception handling, and external I/O.

* All ISAs are generally similar (in principle) and once you are familiar with one, it is easy to learn others
* Some are more difficult to learn – Intel ISA is particularly difficult
* Our examples will be based on MIPS ISA

**Types of ISA**

There are essentially two philosophical approaches to ISA

• Complex Instruction Set Computers (CISC) – E.g. Intel/AMD

• Reduced Instruction Set Computers (RISC) – E.g. SPARC/MIPS/PowerPC

**RISC vs. CISC**

RISC machines have much fewer instructions

1. Hardware is optimized to execute the most common language constructs
2. Does a few things very well
3. Heavy use of pipelining and speculative execution method
4. No hardware support for high-level constructs
5. Harder to write compilers
6. Harder to code complex operations

Our study of ISA will be based MIPS

* MIPS is a RISC processor, with
* Fewer instructions to deal with, and
* Simple instruction format that is
* Uniform and easy to understand
* MIPS processors are used widely in embedded systems (e.g. playstation)
* 32-bit word length
* Contains 32 general-purpose registers, $r0 - $r31
* $r0 is special and always contains the value zero
* Most operations can only be on, and store results to, the registers
* There are special instructions for transferring data to/from memory
* There are a total of about 60 instructions, depending on which variant of MIPS you study
* Super-pipelined architecture divides each instruction into small chunks which
* Allows the clock speed to be increased
* Increases throughput of instructions

**Types of MIPS Instructions**

The instructions of MIPS processors can be divided into 8 broad categories

1. Load/Store Instructions

* Fetch/store items from/to memory
* Variants work on whole or part-words

1. Arithmetic Instruction

* Addition etc. of two variables
* Comparisons

1. Immediate Arithmetic Instructions

* Adding/comparing to constants

1. Shift Instructions

* Bit-rotation

1. Multiply/divide Instructions

* Multiplications and divisions

1. Jump and Branch Instructions

* Used to call subroutines,
* Take branches in the code, and
* Execute loops

1. Coprocessor Instructions

* Send data to (or pass control to) an external coprocessor

1. Special Instructions

We will look at how to use a few of the more common instructions and how they relate to the high-level code that you write.

**An Example:** **Simple Addition**

Consider a simple code fragment

a = b + c;

* There is an instruction add which is useful here
* As a first pass, we will write this as

add a, b, c

* This instruction adds b and c and places the result in a
* This is not quite correct: add instruction does not understand the meaning of a, b, c
* It needs to know where they are
  + Somewhere in the memory
  + We use the notation &a to denote the memory address of a
  + Then we can write

add &a, &b, &c

* This almost correct, but there is another consideration:
* Most MIPS instructions can only access the registers, not the main memory
  + add is one of such instructions
* We need to fetch the data from the main memory first
  + If the variable is accessed a lot, the compiler may choose to store it in one of the registers
  + But this will not normally be the case
  + The instruction we need is:

lw $rx, &a

* This belongs to the category of Load/Store instructions
* Load the word at location &a (i.e. the variable a) and put it into register $rx (where x ranges from 1 to 31)
* There is also the analogous ‘store word’ instruction:

sw $rx, &a

* This takes the contents of $rx and transfers them to memory location &a

//load b to r1

//load c to r2

//add the contents of r1 and r2 and place the result in r3

//store the content of r3 to memory location of a

* So our code for a = b + c; will look like:

lw $r1, &b lw $r2, &c

add $r3, $r1, $r2

sw $r3, &a

**Rmks**

* The actual values of the addresses &a, &b etc. are determined by the operating system at run-time
* What if we have a = b + c – d;?
* We will need another instruction to do the subtraction sub $rx, $ry, $rz
* This subtracts the contents of $rz from the contents of $ry and puts the result in $rx

**Addition and Subtraction**

* a = b + c – d;
* We need to break this down into machine instructions

|  |  |  |  |
| --- | --- | --- | --- |
| lw | $r1, | &b | //load b to r1 |
| lw | $r2, | &c | //load c to r2 |
| lw | $r3, | &d | //load d to r3  //add the contents of r1 and r2 and place the result in r4  //subtract the contents of r3 from the contents r4 and place the result in r5  //store the content of r5 to memory location of a |

sub $r5, $r4, $r3

sw $r5, &a