Lecture Handout

Computer Architecture

Lecture No. 1

Reading Material

Vincent P. Heuring & Harry F. Jordan	Chapter 1
Computer Systems Design and Architecture	1.1, 1.2, 1.3, 1.4, 1.5

Summary

1) Distinction between computer architecture, organization and design
2) Levels of abstraction in digital design
3) Introduction to the course topics
4) Perspectives of different people about computers
5) General operation of a stored program digital computer
6) The Fetch-Execute process
7) Concept of an ISA(Instruction Set Architecture)

Introduction

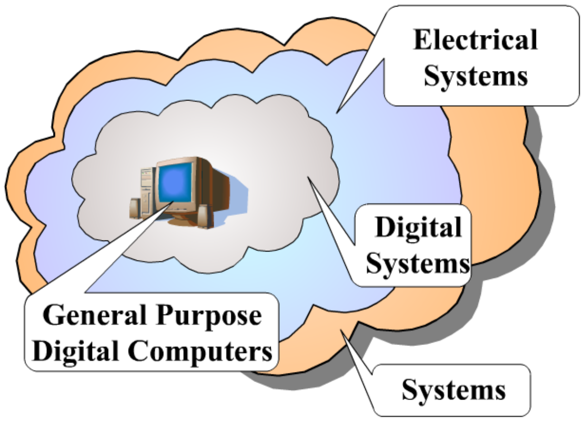
This course is about Computer Architecture. We start by explaining a few key terms.

The General Purpose Digital Computer

How can we define a 'computer'? There are several kinds of devices that can be termed "computers": from desktop machines to the microcontrollers used in appliances such as a microwave oven, from the Abacus to the cluster of tiny chips used in parallel processors, etc. For the purpose of this course, we will use the following definition of a computer:

"an electronic device, operating under the control of instructions stored in its own memory unit, that can accept data (input), process data arithmetically and logically, produce output from the processing, and store the results for future use." [1]

Thus, when we use the term computer, we actually mean a digital computer. There are many digital computers, which have dedicated purposes, for example, a computer used in an automobile that controls the spark



Notion of a System

timing for the engine. This means that when we use the term computer, we actually mean a general-purpose digital computer that can perform a variety of arithmetic and logic tasks.

The Computer as a System

Now we examine the notion of a system, and the place of digital computers in the general universal set of systems. A "system" is a collection of elements, or components, working together on one or more inputs to produce one or more desired outputs.

There are many types of systems in the world. Examples include:

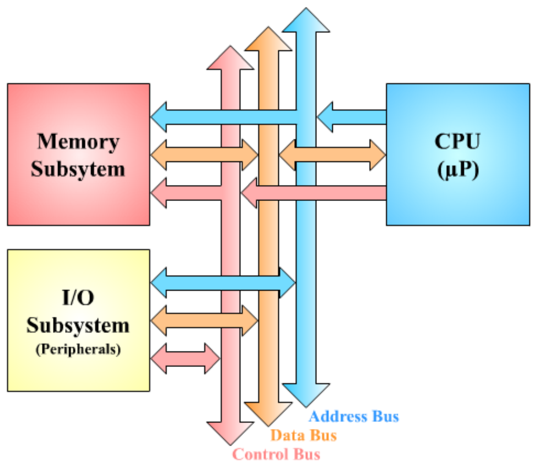
- Chemical systems
- Optical systems
- Biological systems
- Electrical systems
- Mechanical systems, etc.

These are all subsets of the general universal set of "systems". One particular subset of interest is an "electrical system". In case of electrical systems, the inputs as well as the outputs are electrical quantities, namely voltage and current. "Digital systems" are a subset of electrical systems. The inputs and outputs are digital quantities in this case. General-purpose digital computers are a subset of digital systems. We will focus on general-purpose digital computers in this course.

Essential Elements of a General Purpose Digital Computer

The figure shows the block diagram of a modern general-purpose digital computer.

We observe from the diagram that a general-purpose computer has three main components: a memory subsystem, an input/ output subsystem, and a central processing unit. Programs are stored in the memory, the execution of the program instructions takes place in the CPU, and the communication with the external world is achieved through the I/O subsystem (including the peripherals).



Block Diagram of a Computer System

Architecture

Now that we understand the term "computer" in our context, let us focus on the term architecture. The word architecture, as defined in standard dictionaries, is ***"the art or science of building"****, or* ***"a method or style of building"***. [2]

Computer Architecture

This term was first used in 1964 by Amdahl, Blaauw, and Brooks at IBM [3]. They defined it as

"the structure of a computer that a machine language programmer must understand to write a correct (time independent) program for that machine."

By architecture, they meant the programmer visible portion of the instruction set. Thus, a

family of machines of the same architecture should be able to run the same software (instructions). This concept is now so common that it is taken for granted. The x86 architecture is a well-known example.
The study of computer architecture includes

- a study of the structure of a computer
- a study of the instruction set of a computer
- a study of the process of designing a computer

Computer Organization versus Computer Architecture
It is difficult to make a sharp distinction between these two. However, architecture refers to the attributes of a computer that are visible to a programmer, including

- The instruction set
- The number of bits used to represent various data types
- I/O mechanisms
- Memory addressing modes, etc.

On the other hand, organization refers to the operational units of a computer and their interconnections that realize the architectural specifications. These include

- The control signals
- Interfaces between the computer and its peripherals
- Memory technology used, etc.

It is an architectural issue whether a computer will have a specific instruction or not, while it is an organizational issue how that instruction will be implemented.
Computer Architect
We can conclude from the discussion above that a computer architect is a person who designs computers.
Design
Design is defined as
"the process of devising a system, component, or process to meet desired needs."
Most people think of design as a "sketch". This is the usage of the term as a noun. However, the standard engineering usage of the term, as is quite evident from the above definition, is as a verb, i.e., "design is a process". A designer works with a set of stated requirements under a number of constraints to produce the best solution for a given problem. Best may mean a "cost-effective" solution, but not always. Additional or alternate requirements, like efficiency, the client or the designer may impose robustness, etc.. Therefore, design is a decision-making process (often iterative in nature), in which the basic sciences, mathematical concepts and engineering sciences are applied to convert a given set of resources optimally to meet a stated objective.
Knowledge base of a computer architect
There are many people in the world who know how to drive a car; these are the "users" of cars who are familiar with the behavior of a car and how to operate it. In the same way, there are people who can use computers. There are also a number of people in the world who know how to repair a car; these are "automobile technicians". In the same way, we have computer technicians. However, there are a very few people who know how to design a car; these are "automobile designers". In the same way, there are only very few experts in the world who can design computers. In this course, you will learn how to design computers!

These computer design experts are familiar with

- the structure of a computer
- the instruction set of a computer
- the process of designing a computer

as well as few other related things.

At this point, we need to realize that it is not the job of a single person to design a computer from scratch. There are a number of levels of computer design. Domain experts of that particular level carry out the design activity for each level. These levels of abstraction of a digital computer's design are explained below.

Digital Design: Levels of Abstraction

Processor-Memory-Switch level (PMS level)

The highest is the processor-memory-switch level. This is the level at which an architect views the system. It is simply a description of the system components and their interconnections. The components are specified in the form of a block diagram.

Instruction Set Level

The next level is instruction set level. It defines the function of each instruction. The emphasis is on the behavior of the system rather than the hardware structure of the system.

Register Transfer Level

Next to the ISA (instruction set architecture) level is the register transfer level. Hardware structure is visible at this level. In addition to registers, the basic elements at this level are multiplexers, decoders, buses, buffers etc.

The above three levels relate to "system design".

Logic Design Level

The logic design level is also called the gate level. The basic elements at this level are gates and flip-flops. The behavior is less visible, while the hardware structure predominates.

The above level relates to "logic design".

Circuit Level

The key elements at this level are resistors, transistors, capacitors, diodes etc.

Mask Level

The lowest level is mask level dealing with the silicon structures and their layout that implement the system as an integrated circuit.

The above two levels relate to "circuit design".

The focus of this course will be the register transfer level and the instruction set level, although we will also deal with the PMS level and the Logic Design Level.

Objectives of the course

This course will provide the students with an understanding of the various levels of studying computer architecture, with emphasis on instruction set level and register transfer level. They will be able to use basic combinational and sequential building blocks to design larger structures like ALUs (Arithmetic Logic Units), memory subsystems, I/O subsystems etc. It will help them understand the various approaches used to design computer CPUs (Central Processing Units) of the RISC (Reduced Instruction Set Computers) and the CISC (Complex Instruction Set Computers) type, as well as the principles of cache memories.

Important topics to be covered

- Review of computer organization
- Classification of computers and their instructions
- Machine characteristics and performance
- Design of a Simple RISC Computer: the SRC
- Advanced topics in processor design
- Input-output (I/O) subsystems
- Arithmetic Logic Unit implementation
- Memory subsystems

Course Outline

Introduction:

- Distinction between Computer Architecture, Organization and design
- Levels of abstraction in digital design
- Introduction to the course topics

Brief review of computer organization:

- Perspectives of different people about computers
- General operation of a stored program digital computer
- The Fetch – Execute process
- Concept of an ISA

Foundations of Computer Architecture:

- A taxonomy of computers and their instructions
- Instruction set features
- Addressing Modes
- RISC and CISC architectures
- Measures of performance

An example processor: The SRC:

- Introduction to the ISA and instruction formats
- Coding examples and Hand assembly
- Using Behavioral RTL to describe the SRC
- Implementing Register Transfers using Digital Logic Circuits

ISA: Design and Development

- Outline of the thinking process for ISA design
- Introduction to the ISA of the FALCON – A
- Solved examples for FALCON-A
- Learning Aids for the FALCON-A

Other example processors:

- FALCON-E
- EAGLE and Modified EAGLE
- Comparison of the four ISAs

CPU Design:

- The Design Process
- A Uni-Bus implementation for the SRC
- Structural RTL for the SRC instructions
- Logic Design for the 1-Bus SRC
- The Control Unit
- The 2-and 3-Bus Processor Designs
- The Machine Reset
- Machine Exceptions

Term Exam – I

Advanced topics in processor design:

- Pipelining
- Instruction-Level Parallelism
- Microprogramming

Input-output (I/O):

- I/O interface design
- Programmed I/O
- Interrupt driven I/O
- Direct memory access (DMA)

Term Exam – II

Arithmetic Logic Shift Unit (ALSU) implementation:

- Addition, subtraction, multiplication & division for integer unit
- Floating point unit

Memory subsystems:

- Memory organization and design
- Memory hierarchy
- Cache memories
- Virtual memory

References

[1] Shelly G.B., Cashman T.J., Waggoner G.A., Waggoner W.C., Complete Computer Concepts: Microcomputer and Applications. Ferncroft Village Danvers, Massachusetts: Boyd & Fraser, 1992.

[2] Merriam-Webster Online; The Language Centre, May 12, 2003 (http://www.m-w.com/home.htm).

[3] Patterson, D.A. and Hennessy, J.L., Computer Architecture- A Quantitative Approach, 2nd ed., San Francisco, CA: Morgan Kauffman Publishers Inc., 1996.

[4] Heuring V.P. and Jordan H.F., Computer Systems Design and Architecture. Melano Park, CA: Addison Wesley, 1997.

A brief review of Computer Organization

Perceptions of Different People about Computers

There are various perspectives that a computer can take depending on the person viewing

it. For example, the way a child perceives a computer is quite different from how a computer programmer or a designer views it. There are a number of perceptions of the computer, however, for the purpose of understanding the machine, generally the following four views are considered.

The User's View

A user is the person for whom the machine is designed, and who employs it to perform some useful work through application software. This useful work may be composing some reports in word processing software, maintaining credit history in a spreadsheet, or even developing some application software using high-level languages such as C or Java.
The list of "useful work" is not all-inclusive. Children playing games on a computer may argue that playing games is also "useful work", maybe more so than preparing an internal office memo.

At the user's level, one is only concerned with things like speed of the computer, the storage capacity available, and the behavior of the peripheral devices. Besides performance, the user is not involved in the implementation details of the computer, as the internal structure of the machine is made obscure by the operating system interface.

The Programmer's View

By "programmer" we imply machine or assembly language programmer. The machine or the assembly language programmer is responsible for the implementation of software required to execute various commands or sequences of commands (programs) on the computer. Understanding some key terms first will help us better understand this view, the associated tasks, responsibilities and tools of the trade.

Machine Language

Machine language consists of all the primitive instructions that a computer understands and is able to execute. These are strings of 1s and 0s.Machine language is the computer's native language. Commands in the machine language are expressed as strings of 1s and 0s. It is the lowest level language of a computer, and requires no further interpretation.

Instruction Set

A collection of all possible machine language commands that a computer can understand and execute is called its instruction set. Every processor has its own unique instruction set. Therefore, programs written for one processor will generally not run on another processor. This is quite unlike programs written in higher-level languages, which may be portable. Assembly/machine languages are generally unique to the processors on which they are run, because of the differences in computer architecture.

Three ways to list instructions in an instruction set of a computer:

- by function categories
- by an alphabetic ordering of mnemonics
- by an ascending order of op-codes

Assembly Language

Since it is extremely tiring as well as error-prone to work with strings of 1s and 0s for writing entire programs, assembly language is used as a substitute symbolic representation using "English like" key words called mnemonics. A pure assembly language is a language in which each statement produces exactly one machine instruction, i.e. there is a one-to-one correspondence between machine instructions and statements in the assembly language. However, there are a few exceptions to this rule, the

Pentium jump instruction shown in the table below serves as an example.

Example

The table provides us with some assembly statement and the machine language equivalents of the Intel x 86 processor families.

Assembly Language	Machine Language (Binary)	Machine Language (Hex)	Instruction type
add cx, dx	0000 0001 1101 0001	01 D1	Arithmetic
mov alx, 34h	1011 1000 0011 0100 0000 0000	B8 34 00	Data transfer
xor ax, bx	0011 0001 1101 1000	31 D8	Logic
jmp alpha	1110 1011 1111 1100	EB FC	Control

Alpha is a label, and its value will be determined by the position of the jmp instruction in the program and the position of the instruction whose address is alpha. So the second byte in the last instruction can be different for different programs. Hence there is a one-to-many correspondence of the assembly to machine language in this instruction.

Users of Assembly Language

- **The machine designer**
  The designer of a new machine needs to be familiar with the instruction sets of other machines in order to be able to understand the trade-offs implicit in the design of those instruction sets.
- **The compiler writer**
  A compiler is a program that converts programs written in high-level languages to machine language. It is quite evident that a compiler writer must be familiar with the machine language of the processor for which the compiler is being designed. This understanding is crucial for the design of a compiler that produces correct and optimized code.
- **The writer of time or space critical code**
  A complier may not always produce optimal code. Performance goals may force program-specific optimizations in the assembly language.
- **Special purpose or embedded processor programmer**
  Higher-level languages may not be appropriate for programming special purpose or embedded processors that are now in common use in various appliances. This is because the functionality required in such applications is highly specialized. In such a case, assembly language programming is required to implement the required functionality.

Useful tools for assembly language programmers

- **The assembler:**
  Programs written in assembly language require translation to the machine language, and an assembler performs this translation. This conversion process is termed as the assembly process. The assembly process can be done manually as well, but it is very tedious and error-prone.
  An “assembler” that runs on one processor and translates an assembly language program written for another processor into the machine language of the other processor is called a “cross assembler”.

- **The linker:**
  When developing large programs, different people working at the same time can develop separate modules of functionality. These modules can then be 'linked' to

  form a single module that can be loaded and executed. The modularity of programs, that the linking step in assembly language makes possible, provides the same convenience as it does in higher-level languages; namely abstraction and separation of concerns. Once the functionality of a module has been verified for correctness, it can be re-used in any number of other modules. The programmer can focus on other parts of the program. This is the so-called "modular" approach, or the "top-down" approach.
- **The debugger or monitor:**
  Assembly language programs are very lengthy and non-intuitive, hence quite tedious and error-prone. There is also the disadvantage of the absence of an operating system to handle run-time errors that can often crash a system, as opposed to the higher-level language programming, where control is smoothly returned to the operating system. In addition to run-time errors (such as a divide-by-zero error), there are syntax or logical errors.
  A "debugger", also called a "monitor", is a computer program used to aid in detecting these errors in a program. Commonly, debuggers provide functionality such as
  - The display and altering of the contents of memory, CPU registers and flags
  - Disassembly of machine code (translating the machine code back to assembly language)
  - Single stepping and breakpoints that allow the examination of the status of the program and registers at desired points during execution.

  While syntax errors and many logical errors can be detected by using debuggers, the best debugger in the world can catch not every logical error.
- **The development system**
  The development system is a complete set of (hardware and software) tools available to the system developer. It includes
  - Assemblers
  - Linkers and loaders
  - Debuggers
  - Compilers
  - Emulators
  - Hardware-level debuggers
  - Logic analyzers, etc.

Difference between Higher-Level Languages and Assembly Language

Higher-level languages are generally used to develop application software. These high-level programs are then converted to assembly language programs using compilers. So it is the task of a compiler writer to determine the mapping between the high-level-language constructs and assembly language constructs. Generally, there is a "many-to-many" mapping between high-level languages and assembly language constructs. This means that a given HLL construct can generally be represented by many different equivalent assembly language constructs. Alternately, a given assembly language

construct can be represented by many different equivalent HLL constructs.
High-level languages provide various primitive data types, such as integer, Boolean and a string, that a programmer can use. Type checking provides for the verification of proper

usage of these data types. It allows the compiler to determine memory requirements for variables and helping in the detection of bad programming practices.
On the other hand, there is generally no provision for type checking at the machine level, and hence, no provision for type checking in assembly language. The machine only sees strings of bits. Instructions interpret the strings as a type, and it is usually limited to signed or unsigned integers and floating point numbers. A given 32-bit word might be an instruction, an integer, a floating-point number, or 4 ASCII characters. It is the task of the compiler writer to determine how high-level language data types will be implemented using the data types available at the machine level, and how type checking will be implemented.

The Stored Program Concept

This concept is fundamental to all the general-purpose computers today. It states that the program is stored with data in computer's memory, and the computer is able to manipulate it as data. For example, the computer can load the program from disk, move it around in memory, and store it back to the disk.
Even though all computers have unique machine language instruction sets, the 'stored program' concept and the existence of a 'program counter' is common to all machines.
The sequence of instructions to perform some useful task is called a program. All of the digital computers (the general purpose machine defined above) are able to store these sequences of instructions as stored programs. Relevant data is also stored on the computer's secondary memory. These stored programs are treated as data and the computer is able to manipulate them, for example, these can be loaded into the memory for execution and then saved back onto the storage.

General Operation of a Stored Program Computer

The machine language programs are brought into the memory and then executed instruction by instruction. Unless a branch instruction is encountered, the program is executed in sequence. The instruction that is to be executed is fetched from the memory and temporarily stored in a CPU register, called the instruction register (IR). The instruction register holds the instruction while it is decoded and executed by the central processing unit (CPU) of the computer. However, before loading an instruction into the instruction register for execution, the computer needs to know which instruction to load. The program counter (PC), also called the instruction pointer in some texts, is the register that holds the address of the next instruction in memory that is to be executed.
When the execution of an instruction is completed, the contents of the program counter (which is the address of the next instruction) are placed on the address bus. The memory places the instruction on the corresponding address on the data bus. The CPU puts this instruction onto the IR (**instruction register**) to decode and execute. While this instruction is decoded, its length in bytes is determined, and the PC (**program counter**) is incremented by the length, so that the PC will point to the next instruction in the memory. Note that the length of the instruction is not determined in the case of RISC machines, as the instruction length is fixed in these architectures, and so the program counter is always incremented by a fixed number. In case of branch instructions, the

contents of the PC are replaced by the address of the next instruction contained in the present branch instruction, and the current status of the processor is stored in a register called the **Processor Status Word** (PSW). Another name for the PSW is the flag register. It contains the status bits, and control bits corresponding to the state of the processor. Examples of status bits include the sign bit, overflow bit, etc. Examples of control bits include interrupt enable flag, etc. When the execution of this instruction is completed, the contents of the program counter are placed on the address bus, and the entire cycle is repeated. This entire process of reading memory, incrementing the PC, and decoding the instruction is known as the **Fetch and Execute** principle of the stored program computer. This is actually an oversimplified situation. In case of the advanced processors of this age, a lot more is going on than just the simple "fetch and execute" operation, such as pipelining etc. The details of some of these more involved techniques will be studied later on during the course.

The Concept of Instruction Set Architecture (ISA)

Now that we have an understanding of some of the relevant key terms, we revert to the assembly language programmer's perception of the computer. The programmer's view is limited to the set of all the assembly instructions or commands that can the particular computer at hand execute understood/, in addition to the resources that these instructions may help manage. These resources include the memory space and the entire programmer accessible registers. Note that we use the term 'memory space' instead of memory, because not all the memory space has to be filled with memory chips for a particular implementation, but it is still a resource available to the programmer.

This set of instructions or operations and the resources together form the instruction set architecture (ISA). It is the ISA, which serves as an interface between the program and the functional units of a computer, i.e., through which, the computer's resources, are accessed and controlled.

The Computer Architect's View

The computer architect's view is concerned with the design of the entire system as well as ensuring its optimum performance. The optimality is measured against some quantifiable objectives that are set out before the design process begins. These objectives are set on the basis of the functionality required from the machine to be designed. The computer architect

- Designs the ISA for optimum programming utility as well as for optimum performance of implementation
- Designs the hardware for best implementation of instructions that are made available in the ISA to the programmer
- Uses performance measurement tools, such as benchmark programs, to verify that the performance objectives are met by the machine designed
- Balances performance of building blocks such as CPU, memory, I/O devices, and interconnections
- Strives to meet performance goals at the lowest possible cost

Useful tools for the computer architect

Some of the tools available that facilitate the design process are

- Software models, simulators and emulators
- Performance benchmark programs
- Specialized measurement programs

- Data flow and bottleneck analysis
- Subsystem balance analysis
- Parts, manufacturing, and testing cost analysis

The Logic Designer's View

The logic designer is responsible for the design of the machine at the logic gate level. It is the design process at this level that determines whether the computer architect meets cost and performance goals. The computer architect and the logic designer have to work in collaboration to meet the cost and performance objectives of a machine. This is the reason why a single person or a single team may be performing the tasks of system's architectural design as well as the logic design.

Useful Tools for the Logic Designer

Some of the tools available that aid the logic designer in the logic design process are

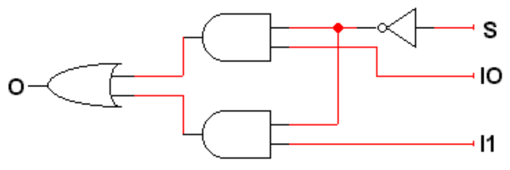
- CAD tools
    - Logic design and simulation packages
    - Printed circuit layout tools
    - IC (integrated circuit) design and layout tools
- Logic analyzers and oscilloscopes
- Hardware development systems

The Concept of the Implementation Domain

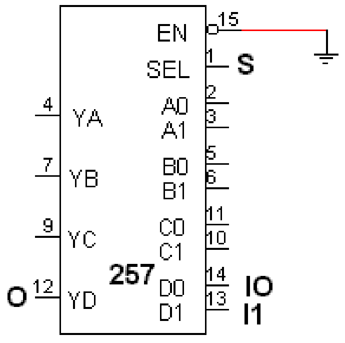
The collection of hardware devices, with which the logic designer works for the digital logic gate implementation and interconnection of the machine, is termed as the implementation domain. The logic gate implementation domain may be

- VLSI (very large scale integration) on silicon
- TTL (transistor-transistor logic) or ECL (emitter-coupled logic) chips
- Gallium arsenide chips
- PLAs (programmable-logic arrays) or sea-of-gates arrays
- Fluidic logic or optical switches

Similarly, the implementation domains used for gate, board and module interconnections are



(a) Abstract view of Boolean logic



(b) TTL implementation domain

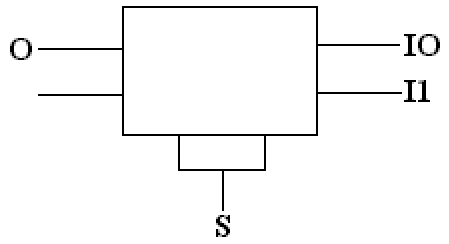
- Poly-silicon lines in ICs
- Conductive traces on a printed circuit board
- Electrical cable
- Optical fiber, etc.

At the lower levels of logic design, the designer is concerned mainly with the functional details represented in a symbolic form. The implementation details are not considered at these lower levels. They only become an issue at higher levels of logic design. An example of a two-to-one multiplexer in various implementation domains will illustrate this point. Figure (a) is the generic logic gate (abstract domain) representation of a 2-to-1 multiplexer.

Figure (b) shows the 2-to-1 multiplexer logic gate implementation

in the domain of TTL (VLSI on Silicon) logic using part number '257, with interconnections in the domain of printed circuit board traces.

Figure (c) is the implementation of the 2-to-1 multiplexer with a fiber optic directional coupler switch, which has an interconnection domain of optical fiber.



(c) Optical switch implementation

Classical logic design versus computer logic design

We have already studied the sequential circuit design concepts in the course on Digital Logic Design, and thus are familiar with the techniques used. However, these traditional techniques for a finite state machine are not very practical when it comes to the design of a computer, in spite of the fact that a computer is a finite state machine. The reason is that employing these techniques is much too complex as the computer can assume hundreds of states.

Sequential Logic Circuit Design

When designing a sequential logic circuit, the problem is first coded in the form of a state diagram. The redundant states may be eliminated, and then the state diagram is translated into the next state table. The minimum number of flip-flops needed to implement the design is calculated by making "state assignments" in terms of the flip-flop "states". A "transition table" is made using the state assignments and the next state table. The flip-flop control characteristics are used to complete a set of "excitation tables". The excitation equations are determined through minimization. The logic circuit can then be drawn to implement the design. A detailed discussion of these steps can be found in most books on Logic Design.

Computer Logic Design

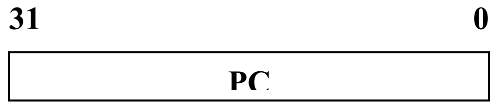
Traditional Finite State Machine (FSM) design techniques are not suitable for the design of computer logic. Since there is a natural separation between the data path and the control path in case of a digital computer, a modular approach can be used in this case.

The data path consists of the storage cells, the arithmetic and logic components and their interconnections. Control path is the circuitry that manages the data path information flow. So considering the behavior first can carry out the design. Then the structure can be considered and dealt with. For this purpose, well-defined logic blocks such as multiplexers, decoders, adders etc. can be used repeatedly.

Two Views of the CPU Program Counter Register

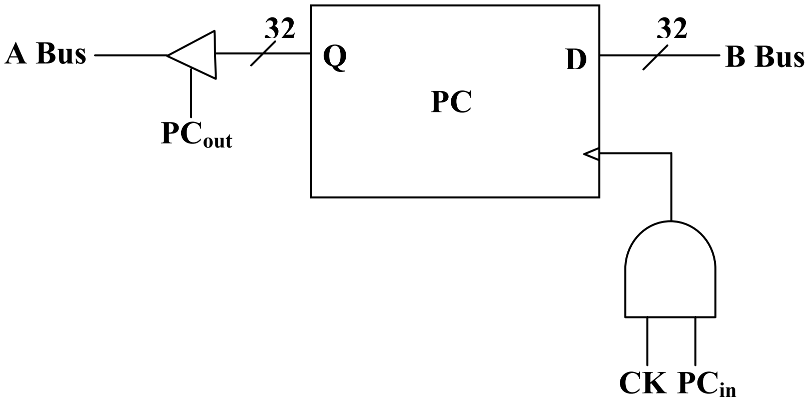
The view of a logic designer is more detailed than that of a programmer. Details of the mechanism used to control the machine are unimportant to the programmer, but of vital importance to the logic designer. This can be illustrated through the following two views of the program counter of a machine.

As shown in figure (a), to a programmer the program counter is just a register, and in this case, of length 32 bits or 4 bytes.



(a) Program Counter: Programmer's view

Figure (b) illustrates the logic designer's view of a 32-bit program counter, implemented as an array of 32 D flip-flops. It shows the contents of the program counter being gated out on 'A bus' (the address bus) by applying a control signal PC_{out}. The contents of the 'B bus' (also the address bus), can be stored in the program counter by asserting the signal PC_{in} on the leading edge of the clock signal CK, thus storing the address of the next instruction in the program counter.



(b) Program Counter: Logic Designer's View