**Computer Organization**

**UNIT-I**

**Bus and Memory Transfers:**

A typical digital computer has many register and paths must be provided to transfer information from one register to another. The number of wires will be excessive if separate lines are used between each register and all other registers in the system. A more efficient scheme for transferring information between registers in a multiple register configuration is a common bus system. A bus structure consists of a set of common lines one for each bit of a register, through which binary information is transferred one at a time .control signals determine which register is selected by the bus during each particular register transfer.

One way of constructing a common bus system is with multiplexers. The multiplexers select the source register whose binary information is then placed on the bus. A shared communication path consists of one and more lines is called bus and data transfer through this bus is called bus transfer. What a data read from memory or stored in memory is called memory transfer. When information is read from memory word is called read operation and data stored in memory is called write operation. The computer has a "bus" where all information travels. It connects all the hardware together. The processor (CPU) calls for a piece of information and it addresses the bus to find that memory location and retrieve it for the operation that needed it. All of the memory cards are hardware plugged into the bus.

A bus, in computing, is a set of physical connections (cables, printed circuits, etc.) which can be shared by multiple hardware components in order to communicate with one another. The purpose of buses is to reduce the number of "pathways" needed for communication between the components, by carrying out all communications over a single data channel. This is why the metaphor of a "data highway" is sometimes used. If only two hardware components communicate over the line, it is called a Hardware port.(Such as a [serial port](http://en.kioskea.net/contents/pc/serie.php3) or [parallel port](http://en.kioskea.net/contents/pc/serie.php3)).



**Micro operations:**

System design uses a modular approach with components at the level of registers, multiplexers, decoders, adders. Gate level design is generally limited to designing the components, although some individual gates may be used to connect larger components. Micro operations are the basic operations that can be performed by a system on data stored in registers. Each micro operation describes a simple operation performed on data in one or more registers.

## Arithmetic Micro operations:

### Types of Micro operations:

* Register Transfer
* Arithmetic (Addition, subtraction,) Data is numeric, and bits with a word are interdependent.
* Logic (AND, OR ...) Data is not numeric, and bits are independent of each other. The same logical operation is applied to each bit in a word in parallel.
* Shift. Data may or may not be numeric. All bits are moved the same number of positions left or right.

###

### Arithmetic Micro operations:

| **Example** | **Description** |
| --- | --- |
| R3 ← R1 + R2 | Addition |
| R3 ← R1 - R2 (R1 + R2' + 1) | Subtraction |
| R2 ← R2' | Complement (really a logic operation) |
| R2 ← -R2 (R2' + 1) | Negation |
| R1 ← R1 + 1 | Increment |
| R1 ← R1 - 1 | Decrement |

Increment and decrement can be done with combinational incrementers and defragmenters, counter registers, or by adding a 1. (Where does the 1 come from?)

Multiply and divide are not often implemented as micro operations due to the amount of time they require. They are usually implemented as a multi-clock-cycle routine of shifts and adds.

## Logic Micro operations:

Logic operations are performed independently on all bits of a word.

 10100010

 ^ 01001010

 ------------

 00000010

As a result, the logic circuit is simply a group of gates working independently on each bit.

To implement a single circuit with many logic functions, simply use a multiplexer.

Logic operations are used heavily in systems programming (device drivers, etc.), cryptography, etc.

* Selective set
* Selective complement
* Selective clear

## Shift Micro operations:

* Arithmetic shift
* Logical shift
* Circular shift

## Arithmetic Logical Shift Unit:

ALU is refer to as Arithmetic Logical shift Unit. The Arithmetic Logical shift Unit can be perform three types of operations these are as Arithmetic operation, Logical operation and Shift operations.

This circuit performs all basic operation. ALU consist of Arithmetic circuit, Logical circuit and Shift circuit. It is consider that every basic circuit can perform some function as Arithmetic circuit perform 8- operations, Logical circuit perform 16- operations and Shift circuit perform 2- operations. The ALU take the input from registers and perform then transfer to the destination register. All the operations are done by ALU is performing in a single clock pulse. But Shift operation is not performing according to clock pulse.

Let us consider a ALU for a single stage with Ith input from register A and B.



## The Function table of ALU is as follows:

|  |  |  |
| --- | --- | --- |
| OPERATION SELECT |  |  |
| s3 | s2 | s1 | s0 | Cin | function | operation |
| 0 | 0 | 0 | 0 | 0 | F=A | TRANSFER |
| 0 | 0 | 0 | 0 | 1 | F=A+1 | INCREMENTER |
| 0 | 0 | 0 | 1 | 0 | F=A+B | ADDER |
| 0 | 0 | 0 | 1 | 1 | F=A+B+1 | ADDER WITH CARRY |
| 0 | 0 | 1 | 0 | 0 | F=A+B' | SUBTRACTION WITH BORROW |
| 0 | 0 | 1 | 0 | 1 | F=A+B'+1 | SUBTRACTION |
| 0 | 0 | 1 | 1 | 0 | F=A-1 | DECREMENTER |
| 0 | 0 | 1 | 1 | 1 | F=A | TRANSFER |
| 0 | 1 | 0 | 0 | \* | F=A^B | AND |
| 0 | 1 | 0 | 1 | \* | F=AvB | OR |
| 0 | 1 | 1 | 0 | \* | F=A XOR B | EX-OR |
| 0 | 1 | 1 | 1 | \* | A' | COMPLIMENT OF A |
| 1 | 0 | \* | \* | \* | F=SHR A | SHIFT RIGHT A TO F |
| 1 | 1 | \* | \* | \* | F=SHL A | SHIFT LEFT A TO F |

Here Ai and Bi are the Ith input from register A and B given both arithmetic circuit as well as logical circuit. Two selection keys S1 and S0 are used for selecting the input to arithmetic and logical circuit. One 4\*1 MUX are used . The output of arithmetic circuit and logical circuit is given to 4\*1 MUX other two inputs are used from shift circuit. Ai-1 is controlled with shift left operation and Ai+1 is controlled with shift right (shr) operation to selection of input S2 and S3 are used for the selection of input to MUX. Ci is the input carry given to arithmetic circuit and Ci+1 is the output carry of arithmetic circuit the output of the ALU is Oi. For n-stage ALU The given circuit replicate in the n-times. I built the ALU from simple logic circuits. I used AND, OR, XOR and NOT gates, and four-bit adders. Subtraction is performed by two's-complement addition, that is, by inverting and adding one to the subtrahend.

**Basic computer organization and design:**

### Computer Instructions:

Computer instructions are the basic components of a machine language program. They are also known as macro operations, since each one is comprised of sequences of micro operations.

Each instruction initiates a sequence of micro operations that fetch operands from registers or memory, possibly perform arithmetic, logic, or shift operations, and store results in registers or memory.

Instructions are encoded as binary instruction codes. Each instruction code contains of a operation code, or opcode, which designates the overall purpose of the instruction (e.g. add, subtract, move, input, etc.). The number of bits allocated for the opcode determined how many different instructions the architecture supports.

In addition to the opcode, many instructions also contain one or more operands, which indicate where in registers or memory the data required for the operation is located. For example, and add instruction requires two operands, and a not instruction requires one.

 15 12 11 6 5 0

 +-----------------------------------+

 | Opcode | Operand | Operand |

 +-----------------------------------+

The opcode and operands are most often encoded as unsigned binary numbers in order to minimize the number of bits used to store them. For example, a 4-bit opcode encoded as a binary number could represent up to 16 different operations.

The control unit is responsible for decoding the opcode and operand bits in the instruction register, and then generating the control signals necessary to drive all other hardware in the CPU to perform the sequence of microoperations that comprise the instruction.

**CPU Block Diagram**



### Stored Program Organization

#### Von Neumann and Harvard Architectures

In the Von Neumann architecture, machine instructions and data are stored in the same RAM during program execution.

A Harvard architecture CPU, in contrast, stored the program and the data in separate memory units. For example, many microcontrollers store the program in Flash memory and the data in traditional, volatile RAM.

#### Operand-based Architecture Classification

Architectures are also classified according to how instructions access memory and process data:

* Memory-to-memory: Most instructions can access memory for any operand. The VAX architecture from Digital Equipment Corporation is an example.
* addl3 x, y, sum # x, y, and sum are memory addresses

* Register-memory: Instructions allow only one operand to be a memory address, while the other(s) must be CPU registers. The x86 architecture is an example.
* movl eax, x
* addl eax, y
* movl sum, eax

* Load-store: Only load and store instructions can access memory. All others must use CPU registers for all operands. The MIPS processor, originally from Digital Equipment Corporation is an example.
* lw $t0, x
* lw $t1, y
* add $t0, $t0, $t1
* sw $t0, sum

* Accumulator-based: One special register, called the accumulator (AC), is an implied operand for most operations. The Zylog Z80 is an example.
* load x # AC ← x
* add y # AC ← AC + y
* store sum # sum ←- AC

#### Designing an Instruction Code

Machine instruction codes may all be the same length (e.g. MIPS processor), or codes for different instructions may be different lengths (e.g. x86, VAX processors).

Suppose all instruction codes of a hypothetical accumulator-based CPU are exactly 16 bits. A simple instruction code format could consist of a 4-bit operation code (opcode) and a 12-bit memory address.

 15 12 11 0

 +-----------------------+

 | Opcode | Address |

 +-----------------------+

This would allow for how many different instructions? How much memory?

Suppose a program contains two variables, x and y. The variable x represents address 0x010 and y represents address 0x012. A segment of the list file, which shows machine code and assembly code side-by-side, might appear as follows:

 0 010 add x

 1 012 sub y

We see that the opcode for add is 000 (0x0) and the opcode for sub is 001 (0x1).

The format above represents memory-reference instructions, which act on an operand from memory and the accumulator. Not all operations require a second operand, so some instructions could act on the accumulator alone. In this case, address bits can be used for other purposes. For example:

 clr # AC = 0

 neg # AC = -AC

 not # AC = AC'

 inc # AC = AC + 1

One or more patterns in the 4-bit opcode can be used to signify that the other 12 bits specify an operation instead of an address. This reduces the number of memory-reference instructions possible, but increases the overall instruction count.

 0XXX add XXX

 1XXX sub XXX

 ...

 F000 clr

 F001 neg

 F002 not

 F003 inc

How many memory-reference instructions can this CPU have?

How many non-memory-reference instructions can this CPU have?

As a second example, suppose a load-store architecture computer has 2-operand instructions, 32 registers, 1 megabyte of byte-addressable memory, 4 addressing modes, 50 register-reference instructions, and 6 load-store instructions. What would the instruction code look like for register-reference instructions? What would the instruction code look like for memory-reference instructions?

Solution: Since there are 50 register-reference instructions, we will need 6 bits for the opcode. (6 bits allows for up to 26 unique opcodes.) With 32 registers, we will need 5 bits to specify each register, so the instruction code format will be 16 bits:

 +--------------------------+

 | opcode | reg1 | reg2 |

 +--------------------------+

 6 5 5

For 6 load-store opcodes, we need 3 bits. 4 addressing modes requires 2 bits to go with the address. A possible instruction code format is as follows:

 +-------------------------------+

 | opcode | reg | mode | address |

 +-------------------------------+

 3 5 4 20

As a third example, suppose a register-memory architecture has 8 registers, a 64k memory capacity, 100 instructions, and 6 addressing modes. Design an instruction code format for memory-reference instructions.

Solution: To represent 100 instructions, we will need 7 bits for the opcode. We'll need 16 bits for a memory address for 64k memory, 3 bits to represent one of 8 registers, and 3 bits to cover all 6 addressing modes for the memory operand. One possible instruction format is shown below. Since this adds up to 19 bits, we would likely use 24 bits for the instruction code to make it fit well into byte-addressable memory. The additional bits could be used to support more opcodes and/or addressing modes.

 +-------------------------------+

 | opcode | reg | mode | address |

 +-------------------------------+

 7 3 3 16

Suppose a direct address in assembly language is represented by a label, as in x below, and an indirect address by a label in parentheses, as in (ptr) below.

 mov x, r3

 add (ptr), r3

If the opcode for mov is 00000001, add is 0000010, the mode bits for direct addressing are 100, and the bits for indirect addressing are 101, the address x is 0x00F0, and ptr is 0x00F2, the instruction codes for the two instructions above would be:

 0000001 011 100 0000000011110000

 0000010 011 101 0000000011110010

Design an instruction code format for a memory-to-memory architecture with 16 registers, a 4 gigabyte memory capacity, 250 instructions, and 16 addressing modes. Assume that there are anywhere from 0 to 3 operands per instruction, and each operand may be in a register or in memory.

#### Some Common Addressing Modes

* Direct: Instruction code contains address of operand
* 0 005 AC = AC + contents of address 5

\* 1 memory-reference beyond fetching instruction

* Immediate: Instruction code contains operand
* 1 005 AC = AC + 5

\* No memory-reference after fetching instruction

* Indirect: Instruction code contains address of address of operand
* 2 005 AC = AC + contents of address stored at address 5

\* 2 memory-references after fetching instruction

Effective address = actual address of the data in memory.

**Effective Address by Addressing Mode**

| **Mode** | **Effective Address** |
| --- | --- |
| Immediate | Address of the instruction itself |
| Direct | Address contained in the instruction code |
| Indirect | Address at the address in the instruction code |

####  Basic Computer Instruction Format:

The Basic Computer has a 16-bit instruction code similar to the examples described above. It supports direct and indirect addressing modes.

How many bits are required to specify the addressing mode?

 15 14 12 11 0

 +------------------+

 | I | OP | ADDRESS |

 +------------------+

 I = 0: direct

 I = 1: indirect

## Timing and Control

All sequential circuits in the Basic Computer CPU are driven by a master clock, with the exception of the INPR register.

At each clock pulse, the control unit sends control signals to control inputs of the bus, the registers, and the ALU.

Control unit design and implementation can be done by two general methods:

* A hardwired control unit is designed from scratch using traditional digital logic design techniques to produce a minimal, optimized circuit. In other words, the control unit is like an ASIC (application-specific integrated circuit).
* A micro programmed control unit is built from some sort of ROM. The desired control signals are simply stored in the ROM, and retrieved in sequence to drive the micro operations needed by a particular instruction.

## Instruction Cycle

In this chapter, we examine the sequences of micro operations that the Basic Computer goes through for each instruction. Here, you should begin to understand how the required control signals for each state of the CPU are determined, and how they are generated by the control unit.

The CPU performs a sequence of micro operations for each instruction. The sequence for each instruction of the Basic Computer can be refined into 4 abstract phases:

1. Fetch instruction
2. Decode
3. Fetch operand
4. Execute

Program execution can be represented as a top-down design:

1. Program execution
	1. Instruction 1
		1. Fetch instruction
		2. Decode
		3. Fetch operand
		4. Execute
	2. Instruction 2
		1. Fetch instruction
		2. Decode
		3. Fetch operand
		4. Execute
	3. Instruction 3 ...

Program execution begins with:

PC ← address of first instruction, SC ← 0

After this, the SC is incremented at each clock cycle until an instruction is completed, and then it is cleared to begin the next instruction. This process repeats until a HLT instruction is executed, or until the power is shut off.

### Instruction Fetch and Decode

The instruction fetch and decode phases are the same for all instructions, so the control functions and microoperations will be independent of the instruction code.

Everything that happens in this phase is driven entirely by timing variables T0, T1 and T2. Hence, all control inputs in the CPU during fetch and decode are functions of these three variables alone.

T0: AR ← PC

T1: IR ← M[AR], PC ← PC + 1

T2: D0-7 ← decoded IR(12-14), AR ← IR(0-11), I ← IR(15)

For every timing cycle, we assume SC ← SC + 1 unless it is stated that SC ← 0.

The operation D0-7 ← decoded IR(12-14) is not a register transfer like most of our microoperations, but is actually an inevitable consequence of loading a value into the IR register. Since the IR outputs 12-14 are directly connected to a decoder, the outputs of that decoder will change as soon as the new values of IR(12-14) propagate through the decoder.

Note that incrementing the PC at time T1 assumes that the next instruction is at the next address. This may not be the case if the current instruction is a branch instruction. However, performing the increment here will save time if the next instruction immediately follows, and will do no harm if it doesn't. The incremented PC value is simply overwritten by branch instructions.

In hardware development, unlike serial software development, it is often advantageous to perform work that may not be necessary. Since we can perform multiple microoperations at the same time, we might was well do everything that might be useful at the earliest possible time.

Likewise, loading AR with the address field from IR at T2 is only useful if the instruction is a memory-reference instruction. We won't know this until T3, but there is no reason to wait since there is no harm in loading AR immediately.

## Memory-Reference Execute Phase

The symbolic representations of the execute phase. For example, the following operation cannot be implemented directly on the Basic Computer:

AC ← AC + M[AR]

Each memory ref instruction is indicated by a unique Di signal.

For the memory-reference execute phase, all control inputs in the CPU are functions of timing signals T4 or later, I, and one of the variables D0 through D6.

The execute phase for memory-reference instructions begins at time T4. The effective address was loaded into AR at time T2 or T3.

Several memory-reference instructions operate on AC and an operand from memory. Since we cannot feed data directly from memory into the ALU

**AND:**

The AND instruction is indicated by signal D0.

D0T4: DR ← M[AR]

D0T5: AC ← AC ^ DR, SC ← 0

**BSA:**

Symbolic notation:

M[AR] ← PC, PC ← AR + 1

Control functions and microoperations:

D5T4: M[AR] ← PC, AR ← AR + 1

D5T5: PC ← AR, SC ← 0

**ISZ:**

ISZ is useful for implementing loops:

 LOOP\_COUNT, DEC -10

 X, DEC 0

 LDA LOOP\_COUNT / Initialize loop counter

 STA X

 LOOP,

 ISZ X / Check loop condition

 BUN LOOP

D6T4: DR ← M [AR]

D6T5: DR ← DR + 1

D6T6: M [AR] ← DR, SC ← 0

D6T6DR (0-15)': PC ← PC + 1

## Input-Output and Interrupt

### Hardware Summary

The Basic Computer I/O consists of a simple terminal with a keyboard and a printer/monitor.

The keyboard is connected serially (1 data wire) to the INPR register. INPR is a shift register capable of shifting in external data from the keyboard one bit at a time. INPR outputs are connected in parallel to the ALU.

 Shift enable

 |

 v

 +-----------+ 1 +-------+

 | Keyboard |---/-->| INPR <|--- serial I/O clock

 +-----------+ +-------+

 |

 / 8

 | | |

 v v v

 +---------------+

 | ALU |

 +---------------+

 |

 / 16

 |

 v

 +---------------+

 | AC <|--- CPU master clock

 +---------------+

How many CPU clock cycles are needed to transfer a character from the keyboard to the INPR register?

Are the clock pulses provided by the CPU master clock?

RS232, USB, Firewire are serial interfaces with their own clock independent of the CPU. ( USB speed is independent of processor speed. )

* RS232: 115,200 kbps (some faster)
* USB: 11 mbps
* USB2: 480 mbps
* FW400: 400 mbps
* FW800: 800 mbps
* USB3: 4.8 gbps

OUTR inputs are connected to the bus in parallel, and the output is connected serially to the terminal. OUTR is another shift register, and the printer/monitor receives an end-bit during each clock pulse.

### I/O Operations

Since input and output devices are not under the full control of the CPU (I/O events are asynchronous), the CPU must somehow be told when an input device has new input ready to send, and an output device is ready to receive more output.

The FGI flip-flop is set to 1 after a new character is shifted into INPR. This is done by the I/O interface, not by the control unit. This is an example of an asynchronous input event (not synchronized with or controlled by the CPU).

The FGI flip-flop must be cleared after transferring the INPR to AC. This must be done as a micro operation controlled by the CU, so we must include it in the CU design.

The FGO flip-flop is set to 1 by the I/O interface after the terminal has finished displaying the last character sent. It must be cleared by the CPU after transferring a character into OUTR.

Since the keyboard controller only sets FGI and the CPU only clears it, a JK flip-flop is convenient:

 +-------+

 Keyboard controller --->| J Q |----->

 | | |

 +--------\-----\ | |

 ) or >----->|> FGI |

 +--------/-----/ | |

 | | |

 CPU-------------------->| K |

 +-------+

**Interrupts**

To alleviate the problems of software polling, a hardware solution is needed.

Analogies to software polling in daily life tend to look rather silly. For example, imagine a teacher is analogous to a CPU, and the students are I/O devices. The students are working asynchronously, as the teacher walks around the room constantly asking each individual student "are you done yet?".

What would be a better approach?

With interrupts, the running program is not responsible for checking the status of I/O devices. Instead, it simply does its own work, and assumes that I/O will take care of itself!

When a device becomes ready, the CPU *hardware* initiates a branch to an I/O subprogram called an *interrupt service routine (ISR)*, which handles the I/O transaction with the device.

An interrupt can occur during *any* instruction cycle as long as interrupts are enabled. When the current instruction completes, the CPU interrupts the flow of the program, executes the ISR, and then resumes the program. The program itself is not involved and is in fact unaware that it has been interrupted.