addressing modes, memory architecture, interrupt and exception handling, and external I/O. An ISA includes a specification of the set of opcodes (machine language), the native commands implemented by a particular CPU design.

1.1 REDUCED INSTRUCTION SET

The acronym **RISC** (pronounced *risk*), for **reduced instruction set computing**, represents a CPU design strategy emphasizing the insight that simplified instructions that "do less" may still provide for higher performance if this simplicity can be utilized to make instructions execute very quickly. Many proposals for a "precise" definition have been attempted, and the term is being slowly replaced by the more descriptive **load-store architecture**. Well known RISC families include Alpha, ARC, ARM, AVR, MIPS, PA-RISC, Power Architecture (including PowerPC), SuperH, and SPARC.

Being an old idea, some aspects attributed to the first RISC-*labeled* designs (around 1975) include the observations that the memory restricted compilers of the time were often unable to take advantage of features intended to facilitate coding, and that complex addressing *inherently* takes many cycles to perform. It was argued that such functions would better be performed by sequences of simpler instructions, if this could yield implementations simple enough to cope with really high frequencies, and small enough to leave room for many registers, factoring out slow memory accesses. Uniform, fixed length instructions with arithmetics restricted to registers were chosen to ease instruction pipelining in these simple designs, with special *load-store* instructions accessing memory.

TYPICAL CHARACTERISTICS OF RISC

For any given level of general performance, a RISC chip will typically have far fewer transistors dedicated to the core logic which originally allowed designers to increase the size of the register set and increase internal parallelism.

Other features, which are typically found in RISC architectures are:

- Uniform instruction format, using a single word with the opcode in the same bit positions in every instruction, demanding less decoding;
- Identical general purpose registers, allowing any register to be used in any context, simplifying compiler design (although normally there are separate floating point registers);
- Simple addressing modes. Complex addressing performed via sequences of arithmetic and/or load-store operations;
- Few data types in hardware, some CISCs have byte string instructions, or support complex numbers; this is so far unlikely to be found on a RISC.

RISC designs are also more likely to feature a Harvard memory model, where the instruction stream and the data stream are conceptually separated; this means that modifying the memory where code is held might not have any effect on the instructions executed by the processor (because the CPU has a separate instruction and data cache), at least until a special synchronization instruction is issued. On the upside, this allows both caches to be accessed simultaneously, which can often improve performance.

5.2 COMPLEX INSTRUCTION SET COMPUTER

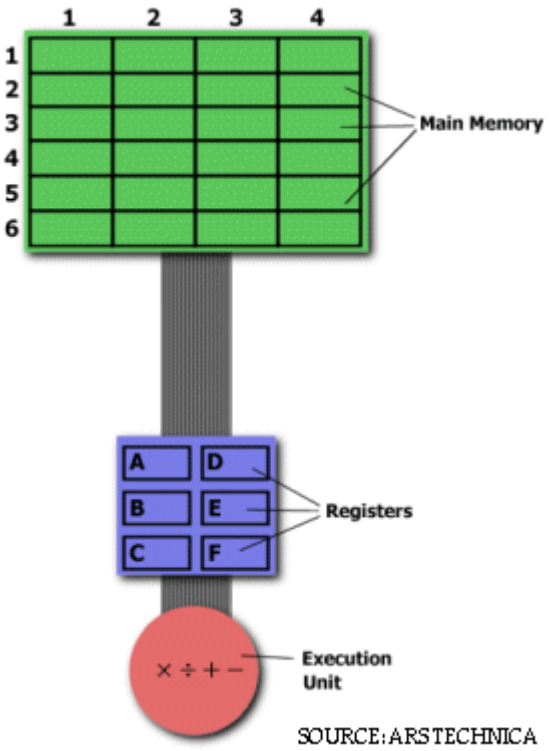
Complex Instruction Set Computers (CISC) have a large instruction set, with hardware support for a wide variety of operations. In scientific, engineering, and mathematical operations with hand coded assembly language (and some business applications with hand coded assembly language), CISC processors usually perform the most work in the shortest time. The primary goal of CISC architecture is to complete a task in as few lines of assembly as possible. This is achieved by building processor hardware that is capable of understanding and executing a series of operations.

RISC VS. CISC

RISC and CISC architecture can be compared by examining the example below;

Multiplying Two Numbers in Memory

On the right is a diagram representing the storage scheme for a generic computer. The main memory is divided into locations numbered from (row) 1: (column) 1 to (row) 6: (column) 4. The execution unit is responsible for carrying out all computations. However, the execution unit can only operate on data that has been loaded into one of the six registers (A, B, C, D, E, or F). Let's say we want to find the product of two numbers - one stored in location 2:3 and another stored in location 5:2 - and then store the product back in the location 2:3.



The CISC Approach

For this particular task, a CISC processor would come prepared with a specific instruction (we'll call it "MULT"). When executed, this instruction loads the two values into separate registers, multiplies the operands in the execution unit, and then stores the product in the appropriate register. Thus, the entire task of multiplying two numbers can be completed with one instruction:

MULT 2:3, 5:2

MULT is what is known as a "complex instruction." It operates directly on the computer's memory banks and does not require the programmer to explicitly call any loading or storing functions. It closely resembles a command in a higher level language. For instance, if we let "a" represent the value of 2:3 and "b" represent the value of 5:2, then this command is identical to the C statement "a = a * b."

One of the primary advantages of this system is that the compiler has to do very little work to translate a high-level language statement into assembly. Because the length of the code is relatively short, very little RAM is required to store instructions. The emphasis is put on building complex instructions directly into the hardware.

THE RISC APPROACH

RISC processors only use simple instructions that can be executed within one clock cycle. Thus, the "MULT" command described above could be divided into three separate commands: "LOAD," which moves data from the memory bank to a register, "PROD," which finds the product of two operands located within the registers, and "STORE," which moves data from a register to the memory banks. In order to perform the exact series of steps described in the CISC approach, a programmer would need to code four lines of assembly:

```
LOAD A, 2:3
LOAD B, 5:2
PROD A, B
STORE 2:3, A
```

At first, this may seem like a much less efficient way of completing the operation. Because there are more lines of code, more RAM is needed to store the assembly level instructions. The compiler must also perform more work to convert a high-level language statement into code of this form.

RISC AND CISC COMPARISON

RISC	**CISC**
Emphasis on hardware	Emphasis on software
Includes multi-clock complex instructions	Single clock, reduced instruction only
Memory-Memory: “LOAD” and “STORE” incorporated in instructions	Register ro register: “LOAD” and “STORE”
Small code sizes, high cycles per second	Low cycles per second, large code sizes
Transistors used for storing complex instructions	Spends more transistors on memory registers

Separating the "LOAD" and "STORE" instructions actually reduces the amount of work that the computer must perform. After a CISC-style "MULT" command is executed, the processor automatically erases the registers. If one of the operands needs to be used for another computation, the processor must re-load the data from the memory bank into a register. In RISC, the operand will remain in the register until another value is loaded in its place.

The CISC approach attempts to minimize the number of instructions per program, sacrificing the number of cycles per instruction. RISC does the opposite, reducing the cycles per instruction at the cost of the number of instructions per program

SECTION TWO

CHAPTER SIX

REGISTERS

Registers are used for storing variables. Because registers are located inside the CPU, they are much faster than memory. Accessing a memory location requires the use of a system bus, so it takes longer time. Accessing data in a register usually takes no time. Therefore, variables should be often kept in registers. From the assembly language point of view, this chapter discusses the 80x86 register sets. The 80x86 processors provide a set of general purpose registers, segment

registers and some special purpose registers. Certain members of the family provide additional registers, although typical applications do not use them. Register sets are very small and most registers have special purpose which limits their use as variables, but they are still an excellent place to store temporary data of calculations.

6.1 GENERAL PURPOSE REGISTERS

8086 CPU has eight 16 bit general purpose registers; each register has its own name:

AX- the accumulator register (divided into AH/AL)

BX- the base register (divided into BH/BL)

CX-the count register (divided into CH/CL)

DX- the data register (divided into DH/DL)

SI- source index register

DI- destination index register

BP- base pointer

SP- stack pointer

Despite the name of a register, it's the programmer who determines the usage for each general purpose register. The size of the above general purpose registers are 16bit, it's something like: 0011000000111001_b

Four general purpose registers (AX, BX, CX, and DX) are made of two separate 8 bit registers, e.g

If AX=0011000000111001_b, then AH=00110000_b and AL=00111001_b. Therefore, when any of the 8 bit registers are modified, 16 bit register is also updated and vice versa. The same thing also applies for the other 3 registers (BX, CX and DX). "H" stands for high and "L" stands for low part.

While you can use many of these registers interchangeably in a computation, many instructions work more efficiently or absolutely require a specific register from this general purpose register group.

A brief explanation of the general purpose registers is given below;

The ax register: This is also called the accumulator. It is where most arithmetic and logical computations take place. Although, most arithmetic and logical operations can be done in other registers, it is often more efficient to use the ax register for such computations

The bx register: This is the base register. It has some special purpose too. It is commonly used to hold indirect addresses.

The cx register: This is the count register. As the name implies, it is used to count off the number of iterations in a loop or to specify the number of characters in a string.

The dx register: This is the data register. This register has two special purposes: It holds the overflow from certain arithmetic operations and it holds I/O addresses when accessing data on the 80x86 I/O bus.

The si and di register are respectively the source index and the destination index register. These registers can be used to indirectly access memory. These registers are also used with the 8086 string instructions when processing character strings.

The bp register: This is the base pointer. It is similar to bx register. It is used to access parameters and local variables in a procedure.

The sp register: This is the stack pointer. It has a very special purpose of maintaining the program stack.

6.2 SEGMENT REGISTERS

8086 CPU has four 16 bit general purpose registers; each register has its own name:

cs register stands for the code segment register

ds register stands for the data segment register

es register stands for the extra segment register

ss register stands foe the stack segment register

Segment registers points at the beginning of a segment in memory. Segments of memory on the 8086 can be no larger than 65,536 bytes long (64K). A brief explanation of the segment registers is given below;

The cs register points at the segment containing the currently executing machine instructions. Despite the 64K segment limitation, 8086 programs can be longer than 64K. This can be possible if multiple code segments are in memory.

The ds register generally points at global variables for the program. Again, one is limited to 65,536 bytes of data in the data segment but the value of ds can always be changed in order to access additional data in other segments.

The es register are often used by programs to gain access to segments when it is difficult or impossible to modify the other segment registers.

The ss register points at the segment containing the 8086 stack. The stack is where the 8086 stores important machine state information, subroutine return addresses, procedure parameters and local variables. In general, the stack segment is not always modified because too many things in the system depend upon it. Although, it is theoretically possible to store data in the segment registers, this is not a good idea.

The segment registers have a very special purpose- pointing at accessible blocks of memory. Any attempt to use the registers for any other purpose may result in considerable grief, especially if intended to move up to better CPU like 80386. Segment registers work together with general purpose register to access any memory value.

e.g If we would like to access memory at the physical address 12345_h(hexadecimal), we should set the DS=1230_h and SI=0045_h. This is good, since this way we can access much more information than with a single register that is limited to 16 bit.

CPU make a calculation of physical address by multiplying the segment register by 10_h and adding a general purpose register to it ($1230_h * 10_h + 45_h = 12345_h$)

The address formed with two registers is called an **effective address.** By default **BX, SI** and **DI** registers work with **DS** segment register; **BP** and **SP** work with **SS** segment register. Other general purpose registers cannot form an effective address. Also, although **BX** can form an effective address, **BH** and **BL** cannot.

10.2 SPECIAL PURPOSE REGISTERS

There are two types of special purpose registers on the 8086 CPU: the instruction pointer (ip) and the flags register. These registers are not accessed the same way other 8086 registers are accessed. Instead, the CPU generally manipulates these registers directly. A brief explanation of the special purpose registers is given below;

The ip register contains the address of the currently executing instruction. It is a 16 bit pointer which provides a pointer into the current code segment. IP works together with the CS segment register and it points to the currently executing instruction.

The flags register is unlike the other registers on the 8086. The other registers hold eight or 16 bit values. The flags register is simply an eclectic collection of on e bit values which help determine the current state of the processor. Flags register is modified automatically by CPU after mathematical operations, this allows to determine the type of the result and to determine the conditions to transfer control to other parts of the program. Generally, the flags register cannot be accessed directly. Although the flags register is 16 bits wide, the 8086 uses only nine of those bits. These bits are explained below;

The carry bits (C): It holds the carry after addition or after subtraction (or the borrow after subtraction). It also indicates an error condition.

The parity bits: it is a count of one in a number expressed as even or odd. It is used as error detection in some circuits.

The Auxiliary carry bit (A): This holds the half-carry after addition or the borrow after subtraction between bit positions 3 and 4 of the results. It is set by a carry out of the lowest nibble (1/2 byte) or a borrow into the lowest nibble.

The zero flag bit (Z): It is set when the result of the operation is zero.

The sign bit (S): This flag holds the arithmetic sign of the result after arithmetic or logic instruction is executed.

The trap flag (T): If the T flag is enabled, the microprocessor interrupts the flow of the program on conditions as indicated by the debug registers and control register. If T=0, the debugging feature is disabled.

The Interrupt flag bit (I): This flag determines whether an external interrupt are disabled or not.

The Direction flag bit (D): This flag determines the direction which string operations are performed. When cleared, string operations proceed from left to right or vice versa.

The overflow flag bit (O): Overflow occurs when signed numbers are added or subtracted. An overflow indicates that the result has exceeded the capacity of the machine.

Of all these flags, four flags are used most times and they are; zero, carry, sign and

CHAPTER SEVEN

ADDRESSING MODES

One approach to processors places an emphasis on flexibility of addressing modes. Some engineers and programmers believe that the real power of a processor lies in its addressing modes. Most addressing modes can be created by combining two or more basic addressing