CPS311 Lecture: Other CPU Architectures: 0,1,2 and 3 address machines
Last revised 9/16/13

Matrials: Addressing Modes Handout

I. Introduction

A. Recall that, at the start of the course, we drew a distinction between computer ARCHITECTURE and computer ORGANIZATION.

1. An architecture is a functional specification of a computer system, generally defined in terms of how the system is seen by an assembly language programmer.

2. An organization a particular way of implementing an architecture.

B. Thus far in this course, we have been studying in detail a particular architecture: that of MIPS. You've also had some experience in lab (and will have more) with a very different architecture: the Z80. Shortly, we will move to a study of basic principles of computer organization. Before we do this, though, we will spend some time looking at the broader range of Von-Neumann style computer architectures that have been implemented. (We will not, at this point, consider alternatives to the basic Von-Neumann architecture that have been proposed.)

1. It turns out that a computer architect faces a number of fundamental choices in designing a CPU architecture. Different architectures represent different combinations of these choice.

ASK CLASS: If you were designing a new CPU architecture, what basic issues would you need to think about? (Recall that a CPU architecture is basically the structure of a machine as seen by an assembly language programmer.)

a. The set of data types that are to be directly supported by the hardware.

b. The size of the logical address space the machine can access.

c. The set of programmer-visible registers that are used to store and manipulate data.

d. The format of individual machine instructions.

e. The set of instructions that are provided in hardware.

f. Provisions for altering program flow based on comparison of values

g. The modes that are provided for generating the address of the operands these instructions work on.

2. We will look at each choice individually.

II. Hardware data types

A. All CPU's provide for the manipulation of binary integers of some specified number of bits (called the word length of the machine).

1. Word lengths for current architectures vary from 8 bits to 64 bits.

a. The earliest microprocessors used an 8 bit word, due to technological reasons. Today's integrated circuit technology permits 32 and 64-bit word lengths, but 8-bit word length microprocessors are still used in applications (e.g. embedded control systems) where a longer word length does not justify the increased cost.

Example: the Z80

b. 16 bit word length CPU's were once fairly common, both in the form of minicomputers (CPU's built from individual chips), and as a technological step up from 8 bit microprocessors), but now if the application calls for more than an 8 bit word the usual step is to a 32 or 64 bit word length (although 16 bit CPU's are still made.)

   i. The chip used in the original IBM PC (Intel 8086/88) used a 16-bit word, and prior to Windows 95 Microsoft's operating systems were based on this architecture, even when running on 32-bit chips. To this day, the need to remain backwards-compatible with the original 16-bit architecture has made the IA32 architecture a bit "wierd".

   ii. Intel has since gone to a 64 bit version of this architecture which offers compability and legacy modes that are backward compatible with the previous 16 and 32 bit versions.

c. From the 1960's onward, mainframe computers have used a 32-bit or (more recently) a 64-bit word size. 32-bit microprocessors became possible during the late 1980's and 64-bit microprocessors in the 1990's.

d. One point of possible confusion: Intel's 32 bit architecture is often called IA32; but IA64 refers to a very different RISC architecture also known as Itanium. The 64 bit version of IA32 is called x86-64. (Actually, this was first developed by AMD, not Intel, and so can also be referred to as AMD64, though Intel doesn't use this term!)

2. Many CPU's actually provide for the manipulation of several sizes of binary integers - often 8, 16, 32 and perhaps 64 bits.

   a. As we have noted, one of these sizes is typically designated as the natural WORD SIZE of the machine, and governs the width of internal registers, data paths etc. (E.g. 32 bits for MIPS I, II).

   b. Frequently, another size (generally the smallest one) is chosen as the basic ADDRESSABLE UNIT, and represents the smallest piece of information that can be transferred to or from a unique address in memory.

      i. On most modern machines, the addressable unit is the byte

      ii. Many older machines, and some modern ones, are only word addressable

      iii. Some machines have been built that allow addressing down to the individual bit.

   c. The drawing of a distinction between the addressable unit and the word size was an important technical innovation that first appeared in the IBM 360 series (early 1960's). It allowed a single architecture to be used for both business applications (which typically work with byte-sized character data) and scientific ones (that typically need long words for arithmetic.)

   d. One interesting issue that arises from the possibility of multiple types of integer is ENDIANNESS.

      i. If the addressable unit is (say) a byte, and the size of an integer is (say) 4 bytes, then an integer occupies locations in memory associated with four successive addresses.

      ii. The universal convention is to treat the FIRST of these addresses as the actual address of the integer - e.g. the

4-byte integer in bytes 1000, 1001, 1002, and 1003 is referred to by the address 1000.

iii. But which bytes of the integer get stored in which location?

- One possibility is to store the MOST significant byte of the integer at the first address, the second most significant byte at the second address .. the least significant byte at the fourth address. This is referred to as BIG-ENDIAN - the "big end" of the number is stored first.

- It is also possible to store the LEAST significant byte of the integer at the first address, the second least significant byte at the second address .. the most significant byte at the fourth address. This is referred to as LITTLE-ENDIAN - the "little end" of the number is stored first.

iv. Example: Consider the hexadecimal integer AABBCCDD, stored at memory location 1000.

- On a big endian machine, we would have:

```
1000 AA
1001 BB
1002 CC
1003 DD
```

- On a little endian machine, we would have

```
1000 DD
1001 CC
1002 BB
1003 AA
```

v. Where endian-ness makes a big difference is when moving data between machines (via a network connection or a file storage medium) that use different conventions.

- Internet protocols call for information to be transmitted over the network in big-endian fashion, regardless of the native format of the processor sending/receiving the data
- Many file formats (both open and proprietary) specify an endianness that they use which must be observed by any machine that reads or writes the the file - even if it must reverse the order of bytes from its "native" convention.

Examples: Adobe Photoshop - big endian
BMP - little endian
GIF - little endian
JPEG - big endian

e. Note that some common data types (boolean, char) are - at the hardware level - simply simply treated as 8 or 16 bit integers. (This means that endianness matters when dealing with unicode characters, though not with ASCII!)

3. Another issue is ALIGNMENT.

a. Some architectures - e.g. many RISCS (including MIPS) require that data units larger than a byte be stored at addresses that are a multiple of the data unit size - e.g. MIPS requires that words (4 bytes) be stored at addresses that are a multiple of 4.

b. Other architectures - e.g. Intel IA32 - impose no such requirement. However, accessing unaligned data in memory is slower, for reasons that will become more apparent when we study memory.

4. Of course, only one size is ESSENTIAL. Other sizes could be

synthesized in software by combining operations on adjacent data items (to get a bigger size) or packing several smaller items into one word. But doing such operations in software requires multiple instructions, and so performance considerations generally dictate offering multiple sizes in hardware.

B. Some CPU's provide for other data types - including some or all of the following:

   1. Binary bit vectors of varying length (not necessarily multiples of the addressable unit.)

   2. Decimal integers, represented using packed binary coded decimal (BCD) or some other scheme.

   3. Floating point numbers. (Sometimes provided for by a separate - and optional - coprocessor chip in microprocessor-based systems.)

   4. Character strings (arbitrary-length strings of bytes).

      Example: MIPS supports only floating point numbers. Both IA32 and the Z80 have BCD arithmetic and hardware support to facilitate character string processing. IA32 also has floating point, which the Z80 does not.

C. The architect's choice of data types to be supported by the hardware has a profound impact on the rest of the architecture.

   1. The tendency in the 1980's was to move toward ever-richer sets of hardware-supported data types. The VAX is really the high water mark of this trend.

   2. RISC's tend to support only the basic integer types (8, 16, 32 and maybe 64 bit) with a floating point coprocessor used for floating point numbers and all other data types managed by software.

III. Size of the Address Space

A. As we have seen, many instructions have to refer to one or more operands contained in memory. One key architectural question is the length, in bits, of the address of an item in memory, because this determines the size of the logical address space the CPU can address, and hence how much memory can be used.

Example: the IBM 360/370 series was originally designed to use a 24 bit address, limiting the address space to 16MB. At the time the architecture was designed (early 1960's), this seemed an exceedingly generous figure; but it has since become a key limitation that has led to a need to significantly modify the architecture.

B. Often, the address size is the same as the word size, because addresses are often calculated using integer arithmetic operations.

Example: MIPS, many others: 32 bit word, 32 bit address

Most 64 bit machines: 64 bit word, 64 address.

C. However, some machines have address sizes smaller than the word size or larger than the word size. The latter requires some tricky mechanisms to allow the formation of an address.

1. Example: the original IBM 360/370 - 32 bit word, but 24 bit address

2. Example: IA32 CPU's used a 32 bit word. They could run in one of two modes:

   "flat addressing": 32 bit address

   "segmented addressing": 48 bit address, computed by using a 32 bit offset plus a 16 bit segment number stored in a special segment register.

   (In either case, though, total physical memory is often much less than 4 GB).

3. Example: x86-64 provides a 64 bit address space in principle, but current implementations are limited to 48 bits (256 TB!). However, this address space is only available when running in full 64-bit mode that is not compatible with the earlier 32 and 16 bit Intel architectures. (When running in a mode that is backward compatible, memory address space is limited to 32 bits).

IV. Register sets

A. Early computers - like the VonNeumann machine - generally provided only a single register that could be directly manipulated by the assembly language programmer - normally called the accumulator.

B. However, as hardware costs dropped it became obvious that having more than one such register would be advantageous.

1. Data that is stored in a register can be accessed much more quickly than data stored in memory, so having multiple registers allows the subset of a program's data items that is currently being manipulated to be accessible more quickly.

2. Modern CPU's typically have anywhere from a half dozen to 32 more or less general purpose registers (plus other special purpose ones.)

3. In fact, some CPU's have been built with as many as 256 registers!

4. However, there are some microprocessors (e.g. the 6502) that have the one accumulator register structure inherited directly from the VonNeumann machine.

C. One question is how functionality should be apportioned among the programmer-visible registers. Here, several approaches can be taken.

1. Multiple registers, each with a specialized function - e.g.:

   - AC (accumulator)
   - MQ (multiplier-quotient: extends the AC for multiply/divide)
   - Base register - one or several
   - Index register - one or several

   Examples: Many micros

   - Z80: A register = accumulator; B..E = temporary storage; IX,IY = index.

   Example: The IA32 architecture is an extension of the 16 bit 80x86 architecture, which in turn is an extension of the 8 bit architecture of the Intel 8080. The result is some specialization of the various registers.

   Example: the EAX is most the most generally useful accumulator. The name is derived from "extended AX", which in turn stands for "extended A", and harkens back to the A register of the 8080! (The 8080 architecture was also the basis for the Z80)

   Example: x86-64 running in 64-bit mode renames the 32 bit registers as 64 bit registers rax, rbx ... and adds 8 general registers known as r8..r15.

2. Multiple general-purpose registers - often designated R0, R1 ....

   Example: Many modern machines including MIPS

3. At the opposite extreme, it is possible to design a machine with NO programmer visible registers per se, by using a stack architecture in which all instructions operate on a stack maintained by the hardware.

   Example: Java Virtual Machine

D. In deciding on a register set architecture, there is an important trade off. When a machine has multiple registers, each instruction must somehow designate which register(s) is/are to be used. Thus, going from one AC to (say) 8 registers adds 3-9 bits to each machine language instruction - depending on how many register operands must be specified.

V. Instruction Formats

A. Different computer architectures differ quite significantly in the format of instructions. Broadly speaking, they can be classified into perhaps four categories, by looking at the format of a typical computational instruction (e.g. ADD). To compare these architectures, we will look at how they implement two HLL statements, assuming all operands are variables in memory:

```
Z := X + Y

Z := (X + Y)/(Q - Z)
```

B. Memory-memory architectures: all operands of a computational instruction are contained in memory.

1. Three address architecture: op destination source1 source2

```
Z = X + Y

        ADD     Z, X, Y

Z = (X + Y)/(Q - Z)

        ADD     T1, X, Y
        SUB     T2, Q, Z
        DIV     Z, T1, T2
```

Example: The above would be valid VAX programs if we used the VAX mnemonics ADDL3, SUBL3, and DIVL3 - except that the VAX orders operands source1, source2, destination!

2. Two address architecture: op destination source

(destination serves as both the second source and the destination)

```
Z = X + Y

        MOV     Z, X
        ADD     Z, Y

Z = (X + Y)/(Q - Z)

        MOV     T1, Q
        SUB     T1, Z
        MOV     Z, X
        ADD     Z, Y
        DIV     Z, T1
```

Example: The above would be valid VAX programs if we used the mnemonics MOVL, ADDL2, SUBL2, and DIVL2 - except that the VAX orders operands source, destination!

3. Since the VAX, memory-memory architectures have gone out of style. (I know of no machines with such an architecture being manufactured today) Note that the VAX was unusual in having both formats

4. Multiple register machines generally allow registers to be used in place of memory locations, as was the case on the VAX.

C. Memory-register architectures: one operand of an instruction is in memory; the other must be in a register. Note: These are often called "one address" architectures.

1. Multiple register machine: op source register (or op register source)

(The designated register serves as both one source and the destination; on some machines (e.g. Intel 80x86), the memory location may also serve as the destination.)
(Multiple register machines generally allow a register to be used instead of the memory operand, as well)

Z = X + Y

```
LOAD    R1, X
ADD     R1, Y
STORE   Z, R1
```

Z = (X + Y)/(Q - Z)

```
LOAD    R1, X
ADD     R1, Y
LOAD    R2, Q
SUB     R2, Z
DIV     R1, R2
STORE   Z, R1
```

Example: An IA32 version of the second program would be

```
MOV     EAX, X
ADD     EAX, Y
MOV     EBX, Q
SUB     EBX, Z
DIV     EAX, EBX
MOV     Z, EAX
```

Example: This is also the format used on the IBM mainframe (360/370) architecture, which is very important historically as well as being still in wide use.

2. Single accumulator machine: op source

(AC serves as both a source and destination)

Z = X + Y

```
LOAD    X
ADD     Y
STORE   Z
```

Z = (X + Y)/(Q - Z)

```
LOAD    Q
SUB     Z
STORE   T1
LOAD    X
ADD     Y
DIV     T1
STORE   Z
```

Example: A very common pattern, including the original Von Neumann machine, many 8-bit microprocessors (including the Z80).

D. Load-store architecture: All operands of a computational instruction must be in registers. Separate load and store instructions are provided for moving a value from memory to a register (load) or from a register to memory (store).

Z = X + Y

```
LOAD    R1, X
LOAD    R2, Y
ADD     R1, R2, R1
STORE   Z, R1
```

Z = (X + Y)/(Q - Z)

```
LOAD    R1, X
LOAD    R2, Y
ADD     R1, R2, R1
LOAD    R2, Q
LOAD    R3, Z
SUB     R2, R2, R3
DIV     R1, R1, R2
STORE   Z, R1
```

Example: RISCs, including MIPS (though MIPS assembly language lists the destination operand of store second)

E. Stack architecture: All operands of a computational instruction are popped from the runtime stack, and the result is pushed back on the stack. Separate push and pop instructions are provided for moving a value from memory to the stack (push), or from the stack to a register (pop).

Z = X + Y

```
PUSH    X
PUSH    Y
ADD
POP     Z
```

```
Z = (X + Y)/(Q - Z)

        PUSH    X
        PUSH    Y
        ADD
        PUSH    Q
        PUSH    Z
        SUB
        DIV
        POP     Z
```

Example: Java Virtual Machine

F. There are trends in terms of program length in the above examples.

1. The example programs for the 3 address machine used the fewest instructions, and those for the load-store and stack machines used the most, with the other architectures in between. (This is a general pattern.)

2. However, the program that is shortest in terms of number of instructions may not be shortest in terms of total number of BITS - because encoding an address in an instruction requires a significant number of bits.

3. You will investigate this further on a homework set.

VI. Instruction Sets

A. Every machine instruction includes an operation code (op-code) that specifies what operation is to be performed. The set of all such possible operations constitutes the INSTRUCTION SET of a machine.

B. Early computers tended to have very limited instruction sets: data movement, basic integer arithmetic operations (+, -, sometimes * and /); integer comparison; bitwise logical and/or; conditional and unconditional branch, subroutine call/return, IO operations and the like.

C. Introducing multiple data types increases the size of the instruction set, since there must be a version of each operation for each (relevant) data type.

D. In the late 1970's and 1980's machines moved in the direction of including some very complex and high-power instructions in the instruction set. The VAX was the high water mark of this trend, with single machine-language instructions for operations like:

1. Character string operations that copy, compare, or search an entire character string of arbitrary length using a single instruction.

2. Direct support for higher-level language constructs like CASE, FOR, etc.

3. Complicated arithmetic operations such as polynomial evaluation.

4. Queue manipulations to or remove an item from a queue.

etc.

E. RISC architectures arose from a questioning of the wisdom of this trend, leading to much simpler instruction sets.

1. The existence of complex instructions imposes a performance penalty on all the instructions in the instruction set - for reasons that will become more evident later. In particular, RISC architecture designers looked carefully at what instructions were actually _used_ by compilers. They found that, in some cases, instructions that were complex to implement (and may have penalized all instructions) were, in fact, rarely if ever used.

2. On the other hand, if the instruction set is kept simple, the door is opened to significant speed-up by the use of pipelining - i.e. executing portions of two or more instructions simultaneously. In particular, having all instructions being the same size facilitates pipelining, by making it possible to determine where one instruction stops and another begins before actually decoding the first one.

3. By using a load-store architecture, all arithmetic and logical instructions can work on a single operand size (e.g. the 32 bit word) - with values being widened or narrowed as appropriate when loading from/storing to memory.

VII. Provisions for Altering Program Flow Based on Comparison of Values

A. All ISA's must provide some mechanism for altering the program flow based on comparing two values, but there are a variety of different ways to accomplish this.

1. First, it is worth noting that only one comparison is really necessary - e.g. less than. As we have already seen, if we have A < B, we can test other conditions as well (e.g. A <= B is ! (B < A); A > B is B < A; A >= B is ! (A < B); A != B is (A < B | B < A); A == B is ! (A < B | B > A)).

2. Moreover, it is only necessary to support either comparing two values, or comparing one value to 0 - e.g. A < B can be tested by seeing if (A - B) < 0.

B. Some ISA's support conditional branches that are based on the value in a register or some comparison between registers.

1. MIPS beq, bne.

2. Some one-accumulator architectures incorporate conditional branches (or skips) based on the value in the accumulator (eg, zero, negative).

C. Many ISA's however, make use of something called CONDITION CODES or FLAGS. A condition code is a bit that is set following an arithmetic or logical operation based on the result of that operation.

1. Example: The Z80 supports condition codes N, Z, C, and V - among others.

   a. S is set to the sign of an operation - and hence is 1 if the result is negative, and 0 if it is not.

   b. Z is set by an operation if the result is zero, and cleared if it is non-zero

   c. C is set by an operation if it produced carry/borrow (unsigned overflow), and cleared if it did not.

   d. V is set by an operation if it produced two's complement overflow, and cleared it it did not.

2. Such architectures often include a "compare" instruction that (in effect) subtracts two values and sets the flags, but does not actually save the result.

3. In such architectures, conditional branch instructions are provided for testing various individual flags or combinations. These instructions merely test the flags - they never alter them. As a result, their outcome is based on the MOST RECENT arithmetic/logic operation to have been performed

   Example: Z80 encoding for

```
if (FOO < BAR)
    S;

LD     A,FOO
LD     B,BAR
CP     A,B
JP     P, FINI // P means "positive" - i.e. S flag not set
Code for S
FINI:
```

4. Of course, in such architectures, it is also possible to base a conditional branch on the result of a computation that needed to be done anyway.

   Example: Z80 encoding for

```
do
{
  ...
  k --;
} while (k != 0)

LOOP:
      ...
      LD A, K
      DEC A
      LD K, A
      JP NZ, LOOP
```

D. MIPS does not use condition codes - in part due to overlap of successive instructions due to pipelining, but also because condition codes make restoring the system state after an interrupt more complicated. (They must be saved like registers to prevent incorrect results if an interrupt occurs between an arithmetic/logic operation and a conditional branch that tests its outcome, which is difficult on a highly pipelined machine where several instructions can be in various stages of execution when an interrupt occurs.)

VIII. Addressing Modes

A. All architectures must provide some instructions that access data in memory - either as part of a computational instruction, or via separate load/store or push/pop instructions. A final architectural issue is how is the address of a memory operand to be specified?

B. The various choices are referred to as ADDRESSING MODES.

1. The VonNeumann machine only provided for direct addressing - the instruction contained the address of the operand.

2. If this is the only way of specifying a memory address, though, then it turns out to be necessary to use SELF-MODIFYING CODE.

a. Example: Consider the following fragment from a C++ program:

```
class Foo
{
    int x;
    ...
};
Foo * p;
...
sum = sum + p -> x;
```

Assuming that x is at the very beginning of a Foo object,
this has to be translated by something like
load p into the AC
add the op-code for add into the AC
store the AC into the point indicate in the program
load sum into the AC
-- this instruction will be created as program is running
store the AC into sum

b. Another example:

```
int x[100];
int i;
...
cin >> i;
sum += x[i];
```

If we assume a byte-addressable machine with 4 byte integers, with x starting at memory address 1000, then successive elements of x occupy addresses 1000, 1004, 1008 .. 1399 (decimal). The ith element of x is at address (1000 + 4*i).

On a one accumulator machine, the operation of adding x[i] to sum translates to

```
load sum into the AC
add x[i] to the AC
store the AC in sum
```

The problem is - what address to we put into the add instruction when writing the program, since the value of i is no known until the program runs (and may even vary if the code in question is contained in a loop that is done multiple times)?

On a computer that only supports direct addressing, we have to code something like:

```
load i into the AC
multiply the AC by 4 (shift left two places)
add base address of x into the AC
add the op-code for add into the AC
store the AC into the point indicate in the program
load sum into the AC
-- this instruction will be created as program is running
store the AC into sum
```

3. Historically, various addressing modes were first developed to address problems like this, then to address other sorts of issues that arise in the program-translation process.

C. The following are commonly-found addressing modes

HANDOUT - not all of the following are in it, just the most important ones

1. Absolute: The instruction contains the ABSOLUTE ADDRESS of the operand

   This is the original addressing mode used on the VonNeumann machine, but still exists on many machines today - at least for some instructions.

   Example (MIPS) j, jal - not supported for other memory reference instructions

2. Memory Indirect (deferred): the instruction contains the address of a MEMORY LOCATION, which in turn contains the ADDRESS OF THE OPERAND.

   Examples: none on any actual machine we have used in the course, but present on some actual machines such as the VAX.

   This mode allows operations like accessing an array element or dereferencing a pointer without using self-modifying code.

   The two examples we developed above could be coded this way

   a. sum += p -> x;

      ```
      load sum into the AC
      add indirect p to the AC
      store the AC into sum
      ```

b. sum += x[i];

```
load i into the AC
multiply the AC by 4 (shift left two places)
add base address of x into the AC
store the AC into the some variable in memory (call it p)
load sum into the AC
add indirect p to the AC
store the AC into sum
```

c. A major disadvantage of supporting this mode is that an extra trip to memory is required to calculate the address of the operand, before actually going to memory to access it. Thus, while CISC architectures may support it, RISC and many CISC architectures (including IA32) do not.

3. Register indirect (register deferred): The ADDRESS of the operand is contained in a REGISTER. (Of course, this is only an option on machines with multiple programmer-visible registers - either general or special purpose)

   Example (MIPS) lw, sw with offset of 0.

   The examples we considered earlier could be coded this way: (where we will denote the register we chose to use for data as the data register and the one we chose to use for addressing as the address register)

   a. sum += p -> x;

   ```
   load sum into the data register
   load p into address register
   add address register indirect to data register
   store the data register into sum
   ```

   b. sum += x[i];

   ```
   load i into the address register
   multiply the address register by 4 (shift left two places)
   add base address of x into the address register
   load sum into the data register
   add address register indirect to data register
   store the data register into sum
   ```

   c. This instruction does not require an extra trip to memory like memory indirect does; however, it may require an extra step to load the pointer variable into a register, as in the first example. As a general rule, all RISC architectures and many CISC architectures (including IA32) offer this option instead of memory indirect.

4. Autoincrement/autodecrement: some architectures support a variant of the register indirect mode in which the register is either incremented after being used as a pointer, or decremented before being used as a pointer, by the size of the operand. [Omitted in HO]

   a. A typical assembler mnemonic is (Rx)+ or -(Rx)

b. This simplifies certain operations, but does not provide any new capability - e.g. the instruction

   something (Rn)+

   is equivalent to

   something register indirect Rn
   increment Rn

c. Provides single-instruction support for stack pushes/pops (examples assume stack grows from high memory to low; sp register points to top of stack)

   e.g. push x move -(sp), x
        pop x   move x, (sp)+

d. Provides single instruction support for accessing array elements in sequence.

   e.g. sum elements of an array (do the following in a loop)

       add (pointer register) +, sum

e. Has found its way into C in the form of the ++ and -- operators applied to pointers (assuming the pointer is kept in a register) - largely because this addressing mode was present on the PDP-11 computer on which C was first implemented, and the language included a facility to take advantage of it for efficiency's sake.

f. RISCs and most CISC's (including IA32) do not provide this mode; instead, one must use a two instruction sequence

5. Register + displacement: The address of the operand is calculated by adding a displacement to a register

   Two uses

   a. Access a component of an object that is not at the start

      Suppose the pointer example we have used were the following:

```
class Foo
{
    int x, y, z;
    ...
};
Foo * p;
...
sum = sum + p -> z;
```

      Now p points to the memory cell holding "x", but the cell holding "z" is 2 words (8 bytes) away.

      This could be done by

load sum into the data register
load p into address register
add 8 to the address register
add address register indirect to data register
store the data register into sum

or by

load sum into the data register
load p into address register
add displaced address register + 8 to data register
store the data register into sum

b. Access an array element specified by a variable index

load i into the address register
multiply address register by 4 (shift left two places)
load sum into the data register
add displaced address register + base address of x to data register
store the data register into sum

c. Example: (MIPS) lw, sw

i. Note that this is the basic addressing mode for MIPS load/store; register indirect is achieved by using this mode with a displacement of 0.

ii. Note that our second example is often not feasible on MIPS because the displacement is limited to 16 bits

d. This mode is commonly available on CISCs as well as RISCs (including IA32)

6. Indexed: [Omitted in HO] some architectures support an addressing mode in which a multiple of some register is added to some base address, where the multiple of the register is the size of the operand - e.g.

Assume we have an array of ints on a byte-addressable machine, where each int occupies 4 bytes, as in our previous examples. Recall that, in this case, to access x[i], we needed code to calculate i * 4 in some address register

If indexed mode were available, we could put the value of i in a register and use indexed mode addressing, where the register is scaled by the size of an element of x[] (4).

Example: a variant of this mode is available on IA32.

7. PC Relative:

   a. Many large programs are made up of multiple modules that are compiled separately, and may also make use of various libraries that are furnished as part of the operating system. When an executable program is linked, the various modules comprising it are generally loaded into successive regions of memory, each beginning where the previous one ends. This implies that, in general, when the compiler is translating a module, it does not know where in memory that module will ultimately be loaded.

      Now consider the problem of translating the following:

```
if (x >= 0)
    y = x;
```

```
load x into some register
compare this register to zero
if it was less than zero, branch to skipstore
store register into y
skipstore:
```

      What address should be placed into the branch instruction i.e. what is the address associated with the label 1$? This is entirely dependent on whether this module is placed in memory by the loader, which the compiler cannot know.

      - One possibility is to require the loader to "fix up" addresses whose value depends on where the module is loaded (so called relocatable addresses)

      - Another approach is to generate position independent code, which works correctly regardless of where in memory this goes.

      In the above example, though the absolute address associated with 1$ is unknown, its relative distance from the branch instruction is fixed.

      In PC relative mode, the instruction would contain the distance between the instruction AFTER the branch instruction and the target, here 1. At run-time, the CPU adds this to the PC to get the actual target address.

   b. Another advantage of PC relative mode is that relative offsets are usually smaller (in bits) than absolute addresses.

   c. Example (MIPS) bne, beq - not supported for other memory reference instructions

      - 16 bit offset

      Note that this mode can also be used for unconditional branching - how?

      ASK

      Use beq $0, $0 - which always branches

D. The following are also considered addressing modes on some machines, though they don't actually address memory. (Depending on how an instruction specifies an addressing mode.)

1. Register: The operand VALUE is contained in a REGISTER (not memory)

   Example (MIPS) "R Format" computational instructions

2. Immediate: The instruction itself CONTAINS the VALUE of the operand

   Example (MIPS) addi, andi, ori, xori etc.

E. Architectures vary in their support for various addressing modes

1. Some architectures use a very flexible and general set of addressing modes - the VAX and the Motrola 68000 series being two examples.

   (On the VAX, any one of the modes listed above can be used with virtually any instruction; the 68000 supports all modes except memory indirect.)

2. Some architectures offer a flexible set of modes, but limit certain modes to use with certain registers. The IA32 and x86-64 are examples of this.

3. Other architectures offer a more restricted set of modes, or specify that specific instructions use specific modes. The latter is the case on MIPS: computational instructions can only use register or immediate mode; jump instructions can only use absolute mode; branch instructions can only use relative mode; and load/store instructions can only use displacement mode, which can give the effect of register indirect by using a displacement of zero, or (for small addresses) of absolute mode by using $0.