CPS311 Lecture: Other Architectures

Last revised June 14, 2021

Objectives:

1. To introduce ISA's in common use today
2. To introduce the difference between RISC and CISC
3. discuss ISA support for various data types.
4. To discuss various register set options.
5. To discuss various instruction formats (3, 2, 1, load-store, and stack(
6. To discuss condition codes.
7. To discuss various addressing mode

Materials:

1. Projectables
2. 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. We frequently use the term Instruction Set Architecture (ISA) to refer to this.

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 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 some of the issues that show up in other Von-Neumann style computer architectures.

C. The following are some ISA's that are in current use.

1. In the 1960's, IBM introduced a family of mainframe computers (one architecture with multiple implementations) known as System/360. Descendants of this ISA are still being produced today as IBM's Z series mainframes for use by large organizations.

   a) A key emphasis in this family is backward compatibility - programs written for an earlier version of the ISA can, in most cases, run on later versions.

      In fact, many programs written for System/360 in the 1960's could be run on Z Series systems being produced today.

   b) Similar computers have been produced by others, such as Univac, Fujitsu, and others.

2. In 1981, IBM released the first personal computer. It used the 16-bit Intel 8086 chip, which was in turn derived from an earlier 8-bit Intel ISA, the 8080. In 1985, this ISA evolved into a 32-bit ISA with the Intel 80386. After further generations, this ISA came to be known as x86, and a 64 bit version commonly known as x86-64.

   a) This is, of course, the ISA used by Windows computers, and has been going back to the precursor MS-DOS system that ran on the original IBM PC.

   b) Since 2005-2006, it has also been the ISA used on Macintoshes.

   c) Linux laptops and PCs use chips that implement this ISA.

   d) Chromebooks use chips that implement this ISA.

   e) This ISA is also widely used in other systems from game systems to server systems.

f) As was the case with System/360, many programs written for the earliest member of this family (8086) can still be run on current implementations of it.

  (1) In fact, backward compatibility has been a major driving force in the design of newer versions of this (and other) ISA's - ensuring that software written for an earlier member of the family can still run on later members.

  (2) In the case of x86 and x86-64, the commitment to backward compatibility with a design based on the 8080 (created in 1974 when what was technically possible was limited) has resulted in an architecture that has a lot of ad-hoc characteristics. According to Patterson and Hennessy (co-inventors of RISC), "this checkered ancestry has led to an architecture that is difficult to explain and impossible to love". (Quoted p 347 of text.)

g) Although the 8086 was first developed by Intel, it has long been the case another company - AMD - also produces chips in this family. This is a consequence of a "second source" requirement imposed by IBM when this chip was first chosen for the IBM PC.

3. The ISA used in most smart phones and tablets (except Windows ones) is known as ARM (Advanced RISC machine). The ancestor of the ARM ISA was first developed in the 1980's, but it has evolved through multiple developments to 32 and 64 bit systems widely used today.

   a) Apple has recently begun a 2 year process of transitioning Macintosh computers from using x86 ISA chips to a new chipset known as M1, which is based on the ARM ISA.

   b) Like the other two architectures, new versions of the ARM ISA offer backward compatibility - especially important between 32 and 64 bit versions.

c) Arm ltd (which owns the patents) licenses other companies to produce implementations, while remaining the developer of new versions of the ISA.

Did you notice that all three of these ISA's were first developed in the 1980's or earlier?

D. We will also talk a bit about the Java Virtual Machine (JVM) - a virtual ISA whose "machine language" is compiled Java bytecodes that reside in `.class` files.

1. There have been some hardware implementations of this ISA, though none in common use.

2. More typically, this virtual machine is implemented by code running on an traditional platform, included as part of the JRE installation for a particular platform.

E. A number of other ISA's are in use in various embedded systems - automobiles, set-top TV boxes, internet routers, etc.

II. RISC vs CISC

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 - largely due to the challenge of creating control hardware.

C. 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, with a view to decreasing what was known as the *semantic gap* between constructs in programming languages and operations supported by the hardware. The idea was to improve performance by needing fewer machine instructions to translate HLL constructs.

The DEC VAX (no longer being manufactured) 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 machine 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.

D. In the 1980's, reduced Instruction Set (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 as we consider control units. In particular, RISC architecture designers looked carefully at what instructions were actually <u>used</u> 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 techniques like 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.

E. In contrast, complex architectures came to be known as complex instruction set (CISC) architectures.

1. Both the IBM mainframe and x86 families are CISCs.

2. MIPS (that we have studied) and ARM are RISCs.

F. Since the 1980's, all new ISA development has used RISC architectures.

III. Hardware data types

A. All ISA'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 ISAs vary from 8 bits to 64 bits.

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) CISCs typically have different instructions for arithmetic operations on different sizes of data (bytes, halfwords, etc.). In RISCs, the arithmetic operations typically work on just one size, with different size load and stores operations for transferring data between memory and registers.

c) Frequently, another size 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.

(1) On most modern machines, the addressable unit is the byte

(2) Many older machines were only word addressable

(3) Some machines have been built that allow addressing down to the individual bit.

(4) 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.)

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

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

b) 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.

c) But which bytes of the integer get stored in which location?

(1) 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 last address. This is referred to as BIG-ENDIAN - the "big end" of the number is stored first.

(2) 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 last address. This is referred to as LITTLE ENDIAN - the "little end" of the number is stored first.

(3) Example: Consider the (32 bit) hexadecimal integer AABBCCDD, stored at memory location 1000.

PROJECT Alternative forms

(a) On a big endian machine, we would have:

1000 AA
1001 BB
1002 CC
1003 DD

(b) On a little endian machine, we would have

1000 DD
1001 CC
1002 BB
1003 AA

(4) ISA's differ in how they treat this issue.

(a) Most older architectures are big-endian, including the IBM mainframe architecture.

(b) The x86 family architectures are little-endian.

(c) A number of architectures are bi-endian - either specified by a hardware bit or a software-settable bit.

(5) 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.

(a) 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.

(b) 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

(6) Note that some common data types (boolean, char) are - at the hardware level - simply treated as 8 or 16 bit integers. Of course, endianness is not an issue for single byte values, but it is for two byte values. (This means that endianness matters when dealing with unicode characters, though not with ASCII!)

4. Another issue is ALIGNMENT.

a) Some architectures (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.x86 - impose no such requirement.

c) However, accessing unaligned data in memory is slower, for reasons that will become more apparent when we study memory.

d) Alignment requirements can impact how a structured data type is stored - and hence can also impact transmission and storage of data.

Example: Consider the following class in C++ or Java:

```
class C {
    int a;
    byte b;
    int c;
    ...
}
```

(1) Storing an object of this class therefore requires 9 bytes.

If we had an array `x` consisting of 2 objects of this class thatwere stored beginning at addresses 1000, the natural way to store it would look like this

```
x[0].a at 1000-1003
x[0].b at 1004
x[0].c at 1005-1008
x[1].a at 1009-1012
x[1].b at 1013
x[1].c at 1014-1017
```

(2) This would not be permissible on an ISA that requires alignment, since most of the 4-byte ints would be at addresses that are not multiples of 4. To fix this, it would be necessary to pad the structure with inaccessible filler bytes - but this would cause a problem if communicating with a machine that does not require alignment!

B. In addition to one or more types of integer, many ISA's provide support for operation on real numbers - often IEEE 754 floats and doubles and sometimes others as well. Of course, the same issues with regard to endianness and alignment arise with real numbers in byte-addressable ISA's.

C. Some ISA's provide for other data types, such as decimal numbers (encoded in BCD) or character strings.

Example: MIPS supports only integers and floating point numbers. Both x86 and the Z80 have BCD arithmetic and hardware support to facilitate character string processing.

D. 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 DEC VAX was really the high water mark of this trend.

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

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 ISA'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 - still being used in embedded systems) that have the one accumulator register structure inherited directly from he 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

a) Example: Many micros - e.g. Z80: A register = accumulator; B..E = temporary storage; X,IY = index.

b) Example: The x86-64 is a 64 bit extension of the 32 bit x86, which in turn is an extension of the 16 bit 8086 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.

Actually, the names of registers hark back to the register names on the 8080 - Intel's 8 bit chip (which was also an ancestor of the Z80)

8080 registers: A, B, C ...
8086 registers AX, BX, CX... plus more not on 8080 - also provide access to 8 bit subsets as A, B, C ...
x86 registers EAX, EBX, ECX ... plus more not on 8086 - also provide access to 8 bit and 16 bit subsets as A, B,C ... or AX, BX, CX ...
x86-64 registers RAX, RBX ... plus more not on x86 - also provide access to 8, 16, and 32 bit subsets as A, B. C ... or AX, BX, CX ... or EAX, EBX, ECX ...

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

   Example: Many modern machines including MIPs.

   a) In some cases - e.g. IBM mainframe ISA, the registers are totally interchangeable with no distinctions between them.

   b) Often, some of the registers can be used as general but will have special ISA functions.

      (1) Example - MIPS - R0 is the constant 0; R31 is used for return address from procedure calls.

      (2) Example - ARM - R15 is the program counter, R14 is used for the return address from procedure calls, and R13 is the stack pointer.

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 tradeoff. 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.

   Note, for example, that almost half the bits in a MIPS R-Format instruction are used to specify registers. (This is a major reason why the IBM mainframe and 32-bit ARM architectures have only 16 general registers.)

V. **Instruction Formats**

A. Different ISA's 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 ISA's, 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)
```

   (PROJECT each format in turn)

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 have been valid VAX programs if we used the VAX mnemonics ADDL3, SUBL3, and DIVL3 - except that the

VAX orders operands source1, source2, destination! However, no current ISA supports having three memory operands in one instruction.

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 have been valid PDP-11 programs and also 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 x86 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

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 - where the A register is implicitly the accumulator).

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 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
```

(Note: the order of pushes is important when using non-commutative operations like subtract or divide. The second item pushed is subtracted from or divided into the first - so the order of pushes corresponds to the order of the original algebraic expression.)

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.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 or cleared following an arithmetic or logical operation based on the result of that operation. (The flag retains its value until the next arithmetic or logical operation is done.)

1. Example: The Z80 supports condition codes S, Z, C, and V, which reside in the F register.

a) S is set to the sign of an arithmetic or logical 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 but rather only sets/clears the condition codes.

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)
   Something;

   LD     A,FOO
   LD     B,BAR
   CP     A,B
   JP     P, FINI // P means "positive"-i.e. S flag clear
   Code for Something
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
     JP NZ, LOOP    // NZ means not zero-i.e. Z flag clear
```

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.)

E. Many other ISA's do, including IBM mainframe, x86, and ARM.

VII.Addressing Modes

A. All ISA's 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 indicated 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];
```

(1) 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).

(2) 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 assembling the program, since the value of i is not known until the program runs (and may even vary if the code in question is contained in a loop that is done multiple times)?

(3) 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 (1000) into the AC
add the op-code for add into the AC
store the AC into the point indicate in the program
add 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 deal with problems like this, then to deal with other sorts of issues that arise in the program-translation process.

C. The following are commonly-found addressing modes

HANDOUT

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 - but not supported for other memory reference instructions

2. The first approach to addressing issues like this was the introduction of memory indirect (deferred) addressing: the instruction contains the address of a MEMORY LOCATION, which in turn contains the ADDRESS OF THE OPERAND.

However, 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 some older CISC architectures supported it, ISA's (both RISC and CISC - including x86) 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 D and the one we chose to use for A)

   a) `sum += p -> x;`

   ```
   load sum into the D
   load p into A
   add A indirect to D
   store the D into sum
   ```

   b) `sum += x[i];`

   ```
   load i into A
   multiply A by 4 (shift left two places)
   add the base address of x (e.g. 1000) into A
   load sum into D
   add A indirect to D
   store the D into sum
   ```

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.

   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 by an appropriate amount
```

c) This 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) This 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) This 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 ISA 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 x86) 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 D
load p A
add value at A displaced by 8 to D
store D into sum
```

b) Access an array element specified by a variable index

```
load i into A
multiply A by 4 (shift left two places)
load sum into D
add value at A displaced by base address of x to D
store D into sum
```

c) Example: (MIPS) lw, sw

(1) 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.

(2) Note that our second example is often not feasible on MIPS because the displacement is limited to 16 bits, so this must be done some other way (e.g. calculating the entire address - base + 4 * value of i and then using register-indirect.)

d) This mode is commonly available on CISCs as well as RISCs (including x86)

6. Indexed: 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 the IBM mainframe and x86 ISA's

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.

b) Now consider the problem of translating the following:

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

translates into:

```
     load x into some register
     compare this register to zero
     if it was less than zero, branch to L1
     store register into y
L1:
```

What address should be placed into the branch instruction i.e. what is the address associated with the label `L1`? This is entirely dependent on whether this module is placed in memory by the loader, which the compiler cannot know.

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

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

c) In the above example, though the absolute address associated with `L1` is unknown, its relative distance from the branch instruction is known and is always the same regardless of where the module is loaded.

In PC relative mode, the instruction would contain the distance between the instruction AFTER the branch instruction and the target, here the length of the store instruction (e.g. 4 on MIPS). At run-time, the CPU adds this to the PC to get the actual target address.

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

e) Example (MIPS) bne, beq - not supported for other memory reference instructions. A 16 bit offset can be used for these, because the target of a branch instruction is generally close to the instruction itself.

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. ISAs vary in their support for various addressing modes

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

   (On the VAX and 68000, any one of the modes listed above can be used with virtually any instruction; for each operand an instruction uses, there is a mode field of 3-4 bits that specifies what addressing mode should be used.)

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

3. Other ISAs 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.