Instruction Set Architecture (ISA - Ch 2)

Dr. Rajib Ranjan Maiti (Mtech and PhD at IIT Kharagpur)
CSIS, BITS-Pilani, Hyderabad

Supporting Procedures in Computer Hardware

- A procedure follow these six steps:
  - 1. Put parameters in a place where the procedure can access them.
  - 2. Transfer control to the procedure.
  - 3. Acquire the storage resources needed for the procedure.
  - 4. Perform the desired task.
  - 5. Put the result value in a place where the calling program can access it.
  - 6. Return control to the point of origin, since a procedure can be called from several points in a program.

- MIPS convention for procedure call :
  - $a0–$a3: four registers to pass arguments
  - $v0–$v1: two registers to return values
  - $ra: one register to hold return address

Supporting Procedures in Computer Hardware

- Lets Consider C procedure:
    - int leaf_example (int g, int h, int i, int j) {
        - int f;
        - f = (g + h) – (i + j);
        - return f;
    - }
- What is the compiled MIPS assembly code?

Supporting Procedures in Computer Hardware

- Lets Consider C procedure:
  - int leaf_example (int g, int h, int i, int j) {
    - int f;
    - f = (g + h) – (i + j);
    - return f;
  - }
- What is the compiled MIPS assembly code?

- The arguments : g, h, i, and j are in $a0, $a1, $a2, and $a3,
- Local variable : f is in $s0
- MIPS Code:

- leaf_example: addi $sp, $sp, –12 #adjust stack
  - sw $t1, 8($sp) # save $t1 for use afterwards
  - sw $t0, 4($sp) # save $t0 for use afterwards
  - sw $s0, 0($sp) # save $s0 for use afterwards
  - add $t0,$a0,$a1 # register $t0 contains g + h
  - add $t1,$a2,$a3 # register $t1 contains i + j
  - sub $s0,$t0,$t1 # f = $t0 – $t1, which is (g + h)–(i + j)
  - add $v0,$s0,$zero # returns f ($v0 = $s0 + 0)
  - lw $s0, 0($sp) # restore $s0 for caller
  - lw $t0, 4($sp) # restore $t0 for caller
  - lw $t1, 8($sp) # restore $t1 for caller
  - addi $sp,$sp,12 # adjust stack to delete 3 items
  - jr $ra # jump back to calling routine

Communicating with People

- **ASCII versus Binary Numbers**
- How much does storage increase if the number 1 billion is represented in ASCII versus a 32-bit integer?
  - Ans. One billion is 1,000,000,000, so it would take 10 ASCII digits, each 8 bits long.
- How MIPS deals with byte?
  - **lb $t0, 0($sp)** # Read byte from memory
  - **sb $t0, 0($gp)** # Write byte to memory

- Consider following C codes
  - void strcpy (char x[], char y[]) {
    - int i;
    - i = 0;
    - while ((x[i] = y[i]) != '\0') /* copy & test byte */
      - i += 1;
  - }
- What is the MIPS assembly code?

MIPS Addressing for I-Type instructions

- Although constants can fit in a 16-bit field,
  - sometimes they are bigger.

- Consider
  - **lui $t0, 255 # load upper immediate - $t0 is register 8:**
- 32 bit format of the instruction

op	rs	rd	Immid.
001111	00000	01000	0000 0000 1111 1111
6 bits	5 bits (unused)	5 bits	16 bits

- Content of $t0 after execution

0000 0000 1111 1111	0000 0000 0000 0000
Upper 16 bits	Lower 16 bits

MIPS Addressing I-Type Instructions

- bne $s0,$s1,Exit # go to Exit if $s0 ≠ $s1

Op	rs	rt	Immid.
5	16	17	Exit
6 bits	5 bits	5 bits	16 bits

MIPS Addressing J-Type Instructions

- **Addressing in Branches and Jumps**
  - j 10000 # go to location 10000

Op	Immid.
2	10000
6 bits	26 bits

Calculating Branch target address

- MIPS assembler code:
  - #variable i is in $s3 and base is in $s6
  - Loop: sll $t1,$s3,2 # Temp reg $t1 = 4 * i
  - add $t1,$t1,$s6 # $t1 = address of save[i]
  - lw $t0,0($t1) # Temp reg $t0 = save[i]
  - bne $t0,$s5, Exit # go to Exit if save[i] ≠ k
  - addi $s3,$s3,1 # i = i + 1
  - j Loop # go to Loop
  - Exit:
- If we assume we place the loop starting at location 80000 in memory, what is the MIPS machine code for this loop?

80000	0	0	19	9	2	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2 **#Exit is +2 away from next instr.**		
80016	8	19	19	1		
80020	2	20000 **#Loop 800000, so, field value = 800000 / 4 = 20000**				
80024	. . .					

Mnemonic	Meaning	Type	Opcode	Funct
add	Add	R	0x00	0x20
addu	Add Unsigned	R	0x00	0x21
and	Bitwise AND	R	0x00	0x24
div	Divide	R	0x00	0x1A
divu	Unsigned Divide	R	0x00	0x1B
jalr	Jump and Link Register	R	0x00	0x09
jr	Jump to Address in Register	R	0x00	0x08
mfhi	Move from HI Register	R	0x00	0x10
mthi	Move to HI Register	R	0x00	0x11
mflo	Move from LO Register	R	0x00	0x12
mtlo	Move to LO Register	R	0x00	0x13
mfc0	Move from Coprocessor 0	R	0x10	NA
mult	Multiply	R	0x00	0x18
multu	Unsigned Multiply	R	0x00	0x19
nor	Bitwise NOR (NOT-OR)	R	0x00	0x27
xor	Bitwise XOR (Exclusive-OR)	R	0x00	0x26
or	Bitwise OR	R	0x00	0x25
slt	Set to 1 if Less Than	R	0x00	0x2A
sltu	Set to 1 if Less Than Unsigned	R	0x00	0x2B
sll	Logical Shift Left	R	0x00	0x00
srl	Logical Shift Right (0-extended)	R	0x00	0x02
sra	Arithmetic Shift Right (sign-extended)	R	0x00	0x03
sub	Subtract	R	0x00	0x22
subu	Unsigned Subtract	R	0x00	0x23

Mnemonic	Meaning	Type	Opcode	Funct
j	Jump to Address	J	0x02	NA
jal	Jump and Link	J	0x03	NA
addi	Add Immediate	I	0x08	NA
addiu	Add Unsigned Immediate	I	0x09	NA
andi	Bitwise AND Immediate	I	0x0C	NA
beq	Branch if Equal	I	0x04	NA
blez	Branch if Less Than or Equal to Zero	I	0x06	NA
bne	Branch if Not Equal	I	0x05	NA
bgtz	Branch on Greater Than Zero	I	0x07	NA
lb	Load Byte	I	0x20	NA
lbu	Load Byte Unsigned	I	0x24	NA
lhu	Load Halfword Unsigned	I	0x25	NA
lui	Load Upper Immediate	I	0x0F	NA
lw	Load Word	I	0x23	NA
ori	Bitwise OR Immediate	I	0x0D	NA
sb	Store Byte	I	0x28	NA
sh	Store Halfword	I	0x29	NA
slti	Set to 1 if Less Than Immediate	I	0x0A	NA
sltiu	Set to 1 if Less Than Unsigned Immediate	I	0x0B	NA
sw	Store Word	I	0x2B	NA

MIPS Assembly/Instruction Formats - Wikibooks, open books for an open world

MIPS Pseudo-Instruction

abs
blt
bgt
ble
neg
negu
not
bge
li
la
move
sge
sgt

Pseudo-Instruction	MIPS Instructions

abs \$1, \$2	addu \$1, \$2, \$0
	bgez \$2, 8 (offset=8 → skip 'sub' instruction)
	sub \$1, \$0, \$2

blt \$8, \$9, label	slt \$1, \$8, \$9 bne \$1, \$0, label
li \$8, 0x3BF20	lui \$at, 0x0003 ori \$8, \$at, 0xBF20
la \$a0,address	lui \$at, 4097 (0x1001 → upper 16 bits of \$at). ori \$a0,\$at,disp
move \$1, \$2	add \$1, \$2, \$0
sge \$1, \$8, \$9	addiu \$9, \$9, -0x01 slt \$1, \$9, \$8

MIPS Assembly/Pseudoinstructions - Wikibooks, open books for an open world

MIPS addressing modes

- Addressing mode
  - Multiple forms of addressing

The MIPS addressing modes

Immediate addressing: the operand is a constant **within the instruction** itself

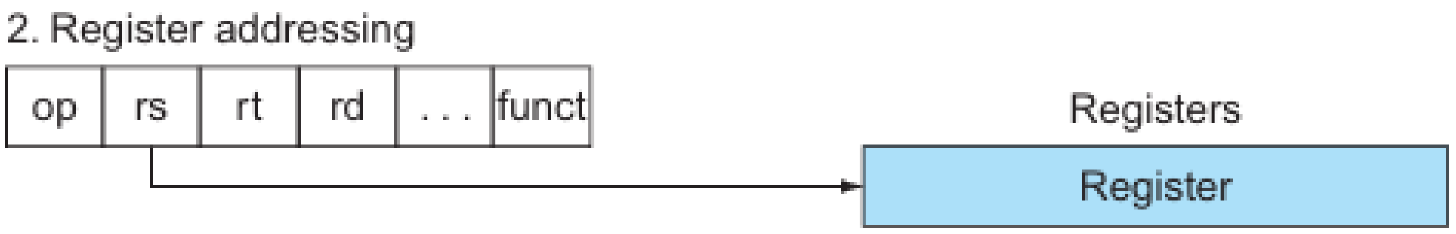
MIPS addressing modes

- Addressing mode
  - Multiple forms of addressing

The MIPS addressing modes

Immediate addressing: the operand is a constant **within the instruction** itself

Register addressing: the operand is **a register**



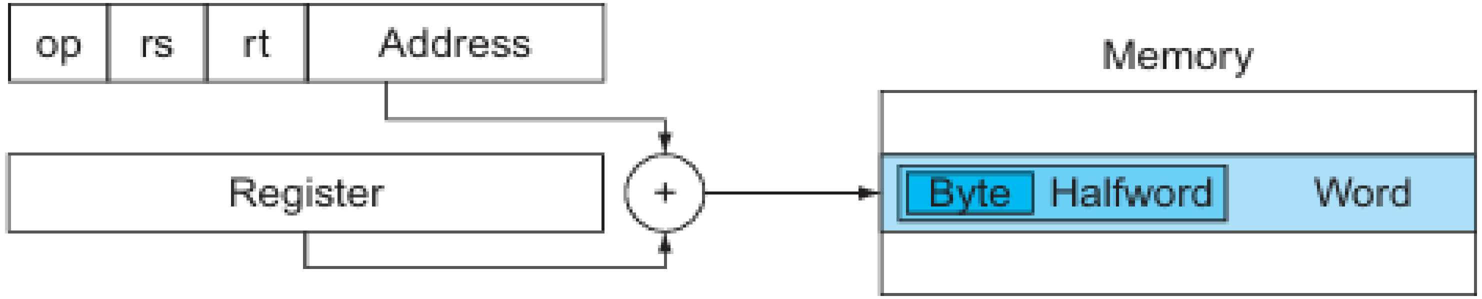
MIPS addressing modes

- Addressing mode
  - Multiple forms of addressing

The MIPS addressing modes

- **Immediate addressing:** the operand is a constant **within the instruction** itself
- **Register addressing:** the operand is **a register**
- **Base or displacement addressing:** the operand is at a memory location of **sum of a register and a constant** in the instruction

3. Base addressing



MIPS addressing modes

- Addressing mode
  - Multiple forms of addressing

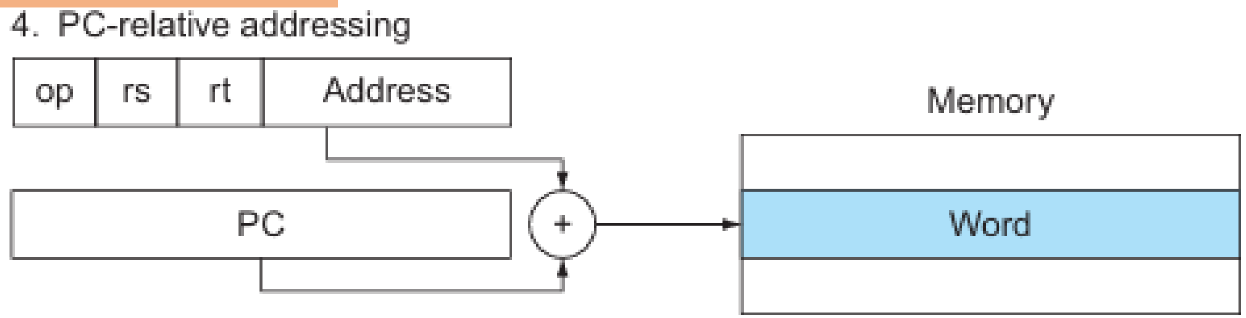
The MIPS addressing modes

Immediate addressing: the operand is a constant **within the instruction** itself

Register addressing: the operand is **a register**

Base or displacement addressing: the operand is at a memory location of **sum of a register and a constant** in the instruction

PC-relative addressing: the branch address is the **sum of the PC and a constant** in the instruction



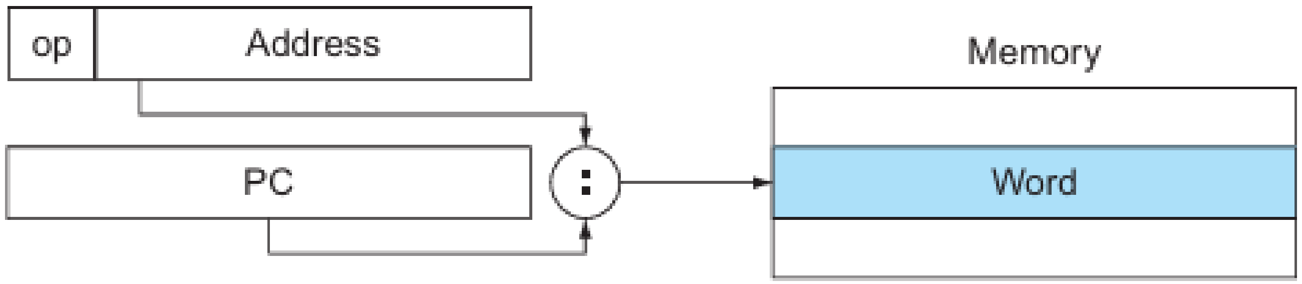
MIPS addressing modes

- Addressing mode
  - Multiple forms of addressing

The MIPS addressing modes

- **Immediate addressing:** the operand is a constant **within the instruction** itself
- **Register addressing:** the operand is **a register**
- **Base or displacement addressing:** the operand is at a memory location of **sum of a register and a constant** in the instruction
- **PC-relative addressing:** the branch address is the **sum of the PC and a constant** in the instruction
- **Pseudodirect addressing:** the jump address is the **26 bits of the instruction** concatenated with the upper bits of the PC

5. Pseudodirect addressing



Parallelism and Instructions: Synchronization

- *Atomic exchange* or *atomic swap*
  - A value in a register ↔ A value in memory
- Build a simple **lock** where
  - Lock = 0 ➔ the lock is free
  - Lock = 1 ➔ the lock is unavailable.
- The value returned from the exchange instruction is
  - 1 if some other processor had already claimed access
  - 0 otherwise.

Parallelism and Instructions: Synchronization

- In MIPS, this pair of instructions includes
  - a special load instruction
    - ***load linked (ll)***
  - a special store
    - ***store conditional (sc)***
- These instructions are used in sequence:
  - if the contents of the memory location specified by the ***ll*** are changed before the ***sc*** to the same address occurs, then ***sc*** fails

Concurrency Problem

- What will be the possible values in *($s0) when the following code executed concurrently by two threads?
  - # *($s0) = 100   -- initial value
  - lw $t0, 0($s0)
  - addi $t0, $t0, 1
  - sw $t0, 0($s0)

- Ans:
  - **101 or 102**
  - 100, 101 or 102
  - 100 or 101
  - 102

Concurrency Problem

- A possible solution
  - Lock (a.k.a. busy wait)
    - Get_lock: # $s0 -> addr of lock
    - addiu $t1, $zero, 1 #t1 = Locked value
    - Loop: lw $t0, 0($s0) #load lock
    - bne $t0, $zero, Loop # loop if locked
    - Lock: sw $t1, 0($s0) # Unlocked, so lock
  - Unlock
    - Unlock: sw $zero, 0($s0)

Any problems with this?

An execution scenario

Thread 1	Thread 2
`addiu $t1,$zero,1`	
`Loop: lw $t0,0($s0)`	
	`addiu $t1,$zero,1`
	`Loop: lw $t0,0($s0)`
`bne $t0,$zero,Loop`	
	`bne $t0,$zero,Loop`
`Lock: sw $t1,0($s0)`	
	`Lock: sw $t1,0($s0)`

Time

Both threads think they have set the lock!
Exclusive access not guaranteed!

A solution using ll and sc (atomic swap)

- ***ll*** returns the initial value, ***sc*** returns 1 only if it succeeds
- For an atomic exchange between 0($s1), i.e., M[$s1 + 0], and $s4:
  - **AGAIN:** addi $t0, $zero, $s4 #$t0 ← $s4
  - ll $t1, 0($s1) #$t1 ← 0($s1) using ***ll***
  - sc $t0, 0($s1) #0($s1) ← $t0 using ***sc***
  - beq $t0, $zero, **AGAIN** #try again if ***sc*** fails
  - add $s4, $zero, $t1 #$s4 ← $t1
- Any time a processor intervenes and modifies the value in memory between the ll and sc instructions,
  - sc returns 0 in $t0 → try again.
- At the end of this sequence,
  - the contents atomically exchanged between $s4 and 0($s1)

Example of Test-and-Set using ll and sc

- In a single atomic operation:
    - **Test :** to see if a memory location is set (contains a 1)
    - **Set :** the memory location to 1, if the location has not contained 1 when tested, i.e., contained a zero
    - **Otherwise :** the **Set** failed, so the program can try again
    - While accessing, no other instruction can modify the memory location, including other Test-and-Set instructions
- Useful for implementing lock operation

Example of Test-and-Set using ll and sc

- MIPS sequence for implementing a T&S at ($s1)
  - Try: addiu $t0, $zero,1 #$t0 ← 1
  - ll $t1, 0($s1) # $t1 ← M[$s1 +0]
  - bne $t1, $zero, Try # if $t1 != 0 then Try again
  - sc $t0, 0($s1) #Otherwise, M[$s1+0] ← $t0
  - beq $t0, $zero,Try #if $t0 == 0 then Try again
  - Locked:
  - critical section
  - sw $zero, 0($s1) #release lock

A C Sort Example to Put It All Together

A C Sort Example to Put It All Together

- The procedure
  - void swap(int v[], int k) {
    - int temp;
    - temp = v[k];
    - v[k] = v[k+1];
    - v[k+1] = temp;
  - }

- To translate from C to assembly language by hand, follow the general steps of:
  - 1. Allocate registers to program variables.
  - 2. Produce code for the body of the procedure.
  - 3. Preserve registers across the procedure invocation.

Steps to get MIPS code

- The variables:
  - The arguments:
    - *v* in $a0
    - *k* in $a1
  - The local variable:
    - *temp* in $t0

- Body of the procedure

```
sll $t1, $a1,2          # reg $t1 = k * 4
add $t1, $a0,$t1        # reg $t1 = v + (k * 4)
                        # reg $t1 has the address of v[k]

lw $t0, 0($t1)          # reg $t0 (temp) = v[k]
lw $t2, 4($t1)          # reg $t2 = v[k + 1]
                        # refers to next element of v

sw $t2, 0($t1)          # v[k] = reg $t2
sw $t0, 4($t1)          # v[k+1] = reg $t0 (temp)
```

- Return of caller

```
jr $ra                  # return to calling routine
```

Arrays versus Pointers

- clear1(int array[], int size) {
  - int i;
  - for (i = 0; i < size; i += 1)
    - array[i] = 0;
- }
- Assumptions:
  - Base Address of Array in $a0
  - Size of array in $a1
  - i in $t0.

- Initiaization
  - move $t0,$zero # i = 0 (register $t0 ← 0)
- To set array[i] to 0
  - loop1: sll $t1,$t0,2 # $t1 = i * 4
  - add $t2,$a0,$t1 # $t2 = address of array[i]
  - sw $zero, 0($t2) # array[i] = 0
  - addi $t0,$t0,1 # i = i + 1
  - slt $t3,$t0,$a1 # $t3 = (i < size)
  - bne $t3,$zero,loop1 # if $t3 !=0 then go to loop1

Arrays versus Pointers: Pointer Version

- clear2(int *array, int size) {
  - int *p;
  - for (p = &array[0]; p < &array[size]; p = p + 1)
    - *p = 0;
- }
- Assumptions:
  - Base Address of Array in $a0
  - Size of array in $a1
  - i in $t0.

- Initiaization:
  - move $t0,$a0 # p = address of array[0]
- To set *p to 0
  - loop2: sw $zero,0($t0) # Memory[p] = 0
  - addi $t0,$t0,4 # p = p + 4
  - sll $t1,$a1,2 # $t1 = size * 4
  - add $t2,$a0,$t1 # $t2 = address of array[size]
  - slt $t3,$t0,$t2 # $t3 = (p<&array[size])
  - bne $t3,$zero,loop2 # if (p<&array[size]) go to loop2

Fallacies and Pitfalls

Fallacies: commonly held misconceptions

- *More powerful instructions mean higher performance.*
  - Find a counterexample or counter argument

- Prefix in Intel x86
  - Part of the power of the Intel x86 is the prefixes that can modify the execution of the following instruction.
  - One prefix can repeat the following instruction until a counter counts down to 0.
  - Thus, to move data in memory, it would seem that the natural instruction sequence is to use move with the repeat prefix to perform 32-bit memory-to-memory moves.

- An alternative method,
  - Use the standard instructions found in all computers,
    - First load the data into the registers
    - Then store the registers back to memory.
  - This second version of this program, with the code replicated to reduce loop overhead, copies at about 1.5 times as fast.
- Another alternative method
  - Use the larger floating-point registers instead of the integer registers of the x86, copies at about 2.0 times as fast than the complex move instruction.

Fallacies: commonly held misconceptions

- *Write in assembly language to obtain the highest performance.*
    - Find a counterexample

- When compilers were poor at register allocation, so programmers use certain hints in high level programming language, like C, to achieve a better performance.
- Today's C compilers generally ignore such hints, because the compiler does a better job at allocation than the programmer does.
- One of the few widely accepted axioms of soft ware engineering is that coding takes longer if you write more lines, and it clearly takes many more lines to write a program in assembly language.
- Moreover, once it is coded, the next danger is that it will become a popular program. Such programs always live longer than expected, meaning that someone will have to update the code over several years and make it work with new releases of operating systems and new models of machines.

Fallacies: commonly held misconceptions

- *The importance of commercial binary compatibility means successful instruction sets don't change.*
  - Find a counterexample

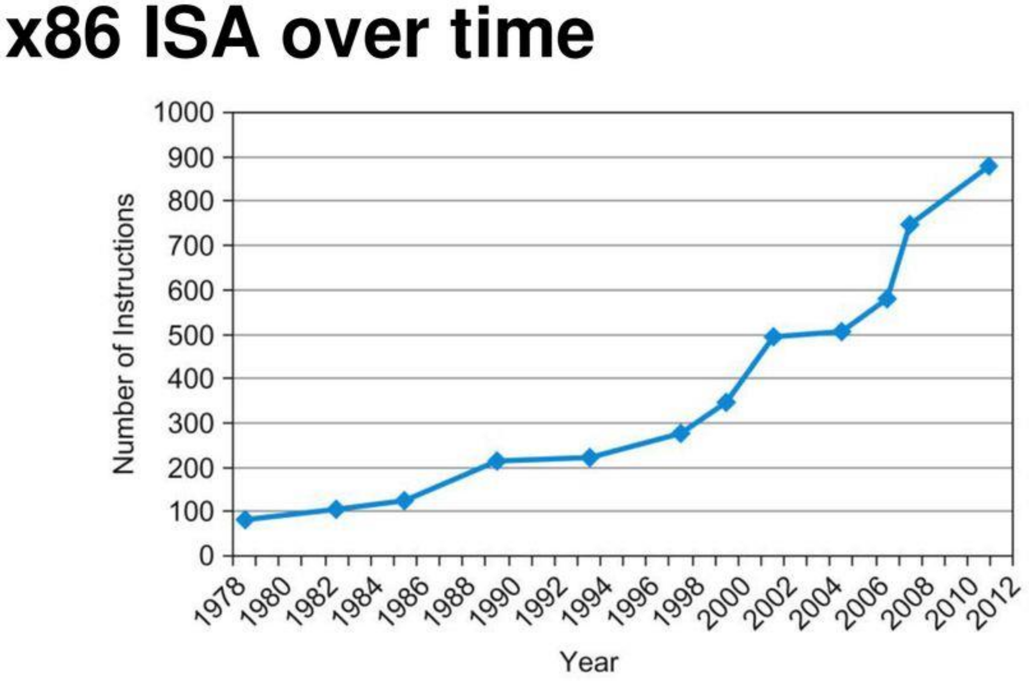


FIGURE 2.43 Growth of x86 instruction set over time. While there is clear technical value to some of these extensions, this rapid change also increases the difficulty for other companies to try to build compatible processors.

Pitfalls: easily made mistakes

- *Forgetting that sequential word addresses in machines with byte addressing do not differ by one.*
    - Because of generalizations of principles that are only true in a limited context

- Many an assembly language programmer has toiled over errors made by assuming that the address of the next word can be found by incrementing the address in a register by one instead of by the word size in bytes.

Pitfalls: easily made mistakes

- *Using a pointer to an automatic variable outside its defining procedure.*
  - Because of generalizations of principles that are only true in a limited context

- pass a result from a procedure that includes a pointer to an array that is local to that procedure
- the memory that contains the local array will be reused as soon as the procedure returns
- Pointers to automatic variables can lead to chaos.

Thank You