Chapter 2

Instructions: Language of the Computer

Computer Architecture (Spring 2016)

Instruction Set

- Instruction = Language of the Machine
  - More primitive than higher level languages
    (ex) no sophisticated control flow
  - Very restrictive (ex) limited number of operands
- Instruction Set = the repertoire of instructions of a computer
- Different *ISA* have different instruction sets
  - But many aspects are found common
- Complexity of the instruction set
  - Early computers had very simple (limited) instruction sets
  - Instruction sets had become more complex ⇒ *CISC*(Complex Instruction Set Computer)
  - Many modern computers also have simple instruction sets ⇒ *RISC*(Reduced Instruction Set Computer)

The MIPS Instruction Set

- Used as the example throughout the book
- Stanford MIPS commercialized by MIPS Technologies
  - Typical of many modern ISAs
- Large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …
- Operands
  - 32 registers
  - 2^{30} memory words
- Operations (refer to the textbook)
  - Arithmetic operations
  - … …

Arithmetic Instructions

- Add and subtract, three operands
  - Two sources and one destination
- An Example
  - C code

```
a = b + c;
```

  - MIPS instruction

```
add a, b, c  # a gets b + c
```

Design Principle 1

- *Simplicity favors regularity*
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost
- Why?
  - It helps make hardware design simple

Arithmetic Instructions: Example

- C code

```
f = (g + h) - (i + j);
```

- Compiled MIPS code

```
add t0, g, h        # temp t0 = g + h
add t1, i, j        # temp t1 = i + j
sub f, t0, t1       # f = t0 - t1
```

Register Operands

- Arithmetic instructions use register operands
- MIPS has a 32 × 32-bit register file
  - Used for frequently accessed data
  - Numbered as 0 to 31
  - 32-bit data called a "word"
- Assembler names
  - $t0, $t1, …, $t9 for temporary values
  - $s0, $s1, …, $s7 for saved variables

Register Operands: Example

- C code:

```
f = (g + h) - (i + j);
```

  - f, g, h, i, j are kept in $s0, $s1, $s2, $s3, and $s4
- Compiled MIPS code:

```
add $t0, $s1, $s2    # temp $t0 = g + h
add $t1, $s3, $s4    # temp $t1 = i + j
sub $s0, $t0, $t1    # f = $t0 - $t1
```

Design Principle 2

- *Smaller is faster*
  - Only 32 registers are provided
  - A small register set brings fast execution

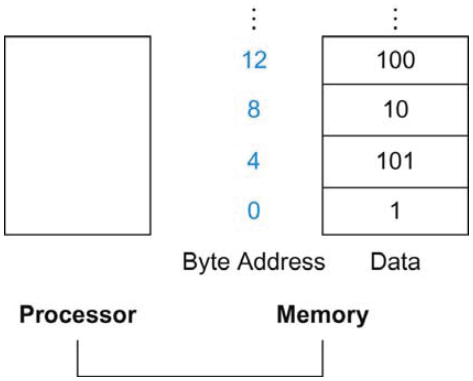
  cf. main memory: millions of locations
- Why?
  - It helps execute individual instructions faster
  - A large register set increases the clock cycle time

Memory Operands

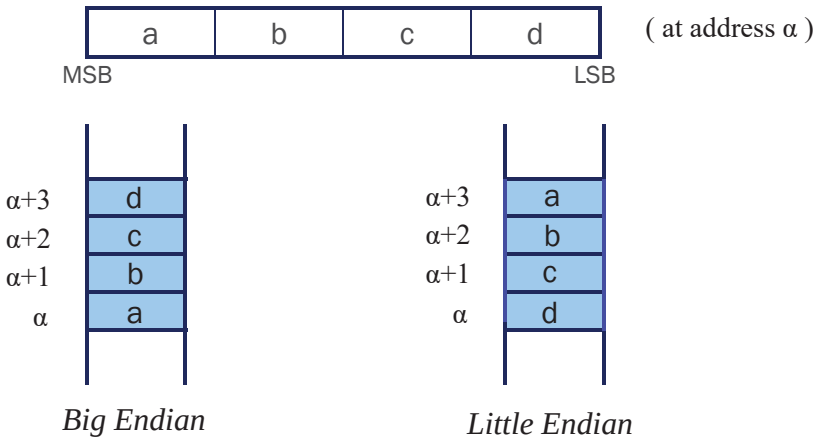
- Main memory used for composite data
  - Arrays, structures, dynamic data
- To execute arithmetic operations in registers, we have to
  - Load values from memory into registers
  - Store result from register to memory
- Memory is *byte addressed*
  - Each address identifies an 8-bit byte
- Words are *aligned* in memory
  - Address must be a multiple of 4
- MIPS is *Big Endian*
  - Most-significant byte is stored at the least (first) address of a word

  cf. *Little Endian*: least-significant byte at least address

Byte Addressing



Big Endian vs. Little Endian



Memory Operands: Example 1

- C code:

```
g = h + A[8];
```

- Compiled MIPS code:

```
lw  $t0, 32($s3)        # load word, A[8]
add $s1, $s2, $t0
```


Memory Operands: Example 2

- C code:

```
A[12] = h + A[8];
```

- Compiled MIPS code:

```
lw  $t0, 32($s3)        # load word
add $t0, $s2, $t0
sw  $t0, 48($s3)        # store word
```

Registers vs. Memory Operand

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
  - More instructions to be executed
- Compiler must use registers for variables as much as possible
  - Only spill to memory for less frequently used variables
  - Register optimization is important!

Immediate Operands

- C code of using a constant 4

```
g = g + 4;
```

- With an immediate operand

```
addi $s3, $s3, 4
```

- Without an immediate operand

```
            lw $t0, AddrConstant4($s1)
            add $s3, $s3, $t0
            … …
somewhere:  4      # AddrConstant4($s1)
```

Utilizing Immediate Operands

- No ‘subtract immediate’ instruction
  - Just use a negative constant

```
addi $s2, $s1, -1
```

- MIPS register 0 ($zero) is reserved to keep the constant 0
  - Hardwired to the value zero
  - Cannot be overwritten
- Utilizing the ‘constant zero’
  - No additional ‘move’ instruction to move between registers

```
add $t2, $s1, $zero
```


Another Design Principle

- *Make the common case fast*
  - Small constants are common
  - Immediate operand avoids a load instruction

```
              lw $t0, AddrConstant4($s1)
              add $s3, $s3, $t0
              … …
somewhere:    4      # AddrConstant4($s1)
```

Unsigned Binary Integers

- Given an n-bit number

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: 0 to $+2^n - 1$
  - Case of 32 bits: 0 to +4,294,967,295
- Example
  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
    $= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$

Signed Binary Integers

- Given an n-bit number

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: -2^{n-1} to $+2^{n-1} - 1$
  - Case of 32 bits:–2,147,483,648 to +2,147,483,647
- Bit 31 is sign bit with 2's complement representation
  - 1 for negative numbers
- Example
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^2 + 0\times2^1 + 0\times2^0$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$

  cf. $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= -(2^{32} - (1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^2 + 0\times2^1 + 0\times2^0))$
    $= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^2 + 0\times2^1 + 0\times2^0$

Sign Extension

- Representing a number using more bits
  - Preserve the numeric value
- Example cases
  - `addi`: extend immediate value
  - `lb`, `lh`: extend loaded byte/halfword
  - `beq`, `bne`: extend the displacement
- Replicate the sign bit to the left
  cf. unsigned values are extended with 0s
- Examples: 8-bit to 16-bit
  - +2: 0000 0010 => 0000 0000 0000 0010
  - –2: 1111 1110 => 1111 1111 1111 1110

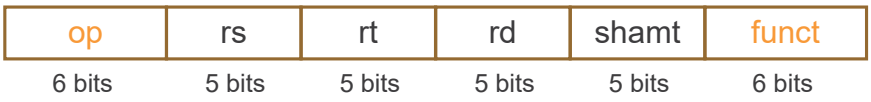
Instruction Formats

- Instructions are kept in the computer
- Instructions are encoded in binary
  - Each piece can be considered as an individual number
- Registers are referred as another number
  - $t0 ~ $t7 are registers 8 ~ 15
  - $t8 ~ $t9 are registers 24 ~ 25
  - $s0 ~ $s7 are registers 16 ~ 23
- *Instruction format*
  - Layout of the instruction
  - Instruction formats encodes *operation code* (*opcode*) and *operands* kept in a register or memory
  - All MIPS instructions are 32 bits long (the same length as data)
  - Regularity!

R-format Instructions

- Instruction fields
  - op: operation code (opcode)
  - rs: first source register number
  - rt: second source register number
  - rd: destination register number
  - shamt: shift amount (00000 for now)
  - funct: function code (extends opcode)

R-format Instructions: Example




```
add $t0, $s1, $s2
```

special	$s1	$s2	$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

$0000\ 0010\ 0011\ 0010\ 0100\ 0000\ 0010\ 0000_2 = 02324020_{16}$

I-format Instructions

- Immediate arithmetic and load/store instructions
  - rt: destination or source register number
  - Constant: -2^{15} to $+2^{15}-1$
  - Address: offset added to base address in rs

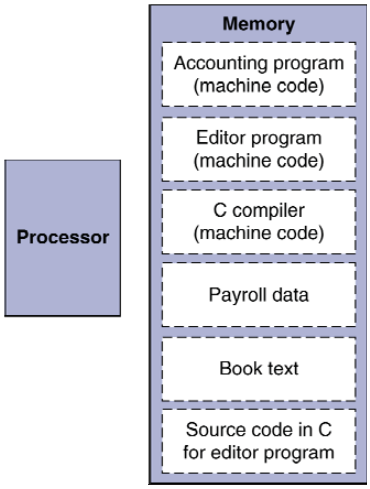
```
addi $s2, $s3, 4
```


```
lw $s2, 4($s3)
```

Design Principle 3

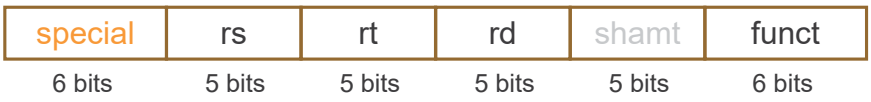
- *Good design demands good compromises*
  - Keep all instructions the same length
  - Allow different formats for different kinds of instructions
  - Different formats may complicate decoding
- Considerations
  - Keep formats as similar as possible

'Stored Program' Computers



- Instructions are represented as numbers
- Programs are stored in memory to be read or written, just like data
- Programs are shipped as files of binary numbers
- *Binary compatibility* allows compiled programs to work on different computers

Logical Instructions



- Instructions for bitwise manipulation

Operation	C	Java	MIPS
Shift left	<<	<<	`sll`
Shift right	>>	>>>	`srl`
Bitwise AND	&	&	`and, andi`
Bitwise OR	|	|	`or, ori`
Bitwise NOT	~	~	`nor`

- Useful for extracting and inserting groups of bits in a word

AND Instruction

- Useful to mask bits in a word
  - Select some bits, clear others to 0

```
and $t0, $t1, $t2
```

$t2	0000 0000 0000 0000 0000 1101 1100 0000
$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	0000 0000 0000 0000 0000 1100 0000 0000

OR Instruction

- Useful to include bits in a word
  - Set some bits to 1, leave others unchanged

```
or $t0, $t1, $t2
```

$t2	0000 0000 0000 0000 0000 1101 1100 0000
$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	0000 0000 0000 0000 0011 1100 0000 0000

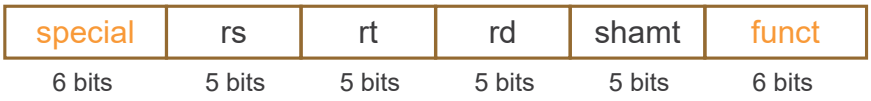
NOT Instruction

- Useful to invert bits in a word
  - Change 0 to 1, and 1 to 0
- MIPS has 3-operand NOR instruction
  - a NOR b == NOT (a OR b)
  - a NOR 0 == NOT (a OR 0) = NOT a

```
nor $t0, $t1, $zero
```

$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	1111 1111 1111 1111 1100 0011 1111 1111

Shift Instructions



- Shift amout (shamt)
  - Specifies how many positions to shift
- Shift left logical
  - Shift left and fill with 0 bits
  - `sll` by *i* bits multiplies by 2^i
- Shift right logical
  - Shift right and fill with 0 bits
  - `srl` by *i* bits divides by 2^i (unsigned only)

Branch Instruction & Jump Instruction

- Moves the control to another location
- Conditional operation (control transfer)
  - Depends on the condition
  - if (rs == rt) branch to instruction labeled L1

```
beq rs, rt, L1
```

  - if (rs != rt) branch to instruction labeled L1

```
bne rs, rt, L1
```

- Unconditional operation (control transfer)
  - unconditional jump to instruction labeled L1

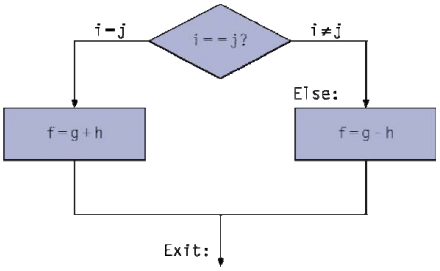
```
j L1
```


Compiling If Statements

- C code

```
if (i==j) f = g+h;
else f = g-h;
```

- Register usage
  - f,g,h → $s0, $s1 , $s2
  - i,j → $s3, $s4



- Compiled MIPS code

```
      bne $s3, $s4, Else   # i != j ?
      add $s0, $s1, $s2    # f = g-h
      j   Exit
Else: sub $s0, $s1, $s2    # f = g+h
Exit: …
```

Compiling Loop Statements

- C code

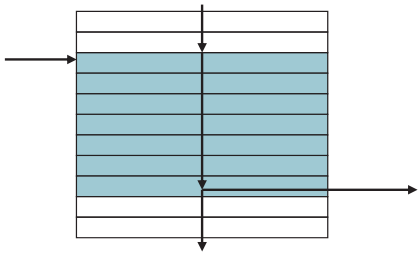
```
while (save[i] == k) i += 1;
```

- Compiled MIPS code:

```
Loop:  sll  $t1, $s3, 2
       add  $t1, $t1, $s6
       lw   $t0, 0($t1)
       bne  $t0, $s5, Exit
       addi $s3, $s3, 1
       j    Loop
Exit: …
```


Basic Blocks

- A *basic block* is a sequence of instructions with
  - No embedded branches (except at end)
  - No branch targets (except at beginning)
- A compiler breaks the program into the basic blocks



- An advanced processor can utilize the basic block for optimization purpose

Why Limited Branch Instructions?

- Both `beq` and `bne` are the common case
- Why not `blt`, `bge`, etc?
- Hardware for <, ≥, … slower than =, ≠
  - Combining with branch involves more work per instruction, requiring a slower clock
  - All instructions penalized!
- This is a good design compromise

Test Instructions

- Test for equality
  - Set result to 1 if a condition is true
  - Otherwise, set to 0
  - if (`rs` < `rt`) `rd` = 1; else `rd` = 0;

```
slt rd, rs, rt
```

  - if (`rs` < constant) `rt` = 1; else `rt` = 0;

```
slti rt, rs, constant
```

- Typical example of using the test instruction

```
slt $t0, $s1, $s2   # if ($s1 < $s2)
bne $t0, $zero, L   #    branch to L
```

Signed vs. *Unsigned* Test Instructions

- Signed comparison: `slt`, `slti`
- Unsigned comparison: `sltu`, `sltui`
- Example
  - `$s0 = 1111 1111 1111 1111 1111 1111 1111 1111`
  - `$s1 = 0000 0000 0000 0000 0000 0000 0000 0001`
  - `slt $t0, $s0, $s1` assigns 1 to `$t0`
  - `sltu $t0, $s0, $s1` assigns 0 to `$t0`

Supporting Procedures in Hardware

- Six steps to support procedures
  1. Place parameters in a place where the procedure can access them (`$a0-$a3`)
  2. Transfer control to the procedure (`jal ProcedureAddress`)
  3. Acquire the storage resources needed for the procedure
  4. Perform the desired task
  5. Release the acquired storage resources
  6. Place the result value in a place where the calling program can access it (`$v0-$v1`)
  7. Return control to the point of origin (`jr $ra`)

Register Usage Convention

Registers	Usage	Remark
$a0-$a3	Arguments	
$v0, $v1	Result values	
$t0-$t9	Temporaries	Can be overwritten by called
$s0-$s7	Saved	Must be saved/restored by callee
$gp	Global pointer for static data	
$sp	Stack pointer	
$fp	Frame pointer	
$ra	Return address	

Procedure Call Instructions

- Procedure call: jump and link

```
jal ProcedureLabel
```

  - Puts the address of following instruction (program counter) in `$ra`
  - Jumps to target address

- Procedure return: jump register

```
jr $ra
```

  - Copies `$ra` to program counter
  - Can also be used for computed jumps (ex) case/switch statements

Leaf Procedure Example

- C code

```
int leaf_example (int g, h, i, j)
{   int f;
    f = (g + h) - (i + j);
    return f;
}
```

- Register usage
  - Arguments g, …, j in $a0, …, $a3
  - f in $s0 (hence, need to save $s0 on stack)
  - Result in $v0

Leaf Procedure Example

- MIPS code

```
leaf_example:
    addi $sp, $sp, -4
    sw   $s0, 0($sp)
    add  $t0, $a0, $a1
    add  $t1, $a2, $a3
    sub  $s0, $t0, $t1
    add  $v0, $s0, $zero
    lw   $s0, 0($sp)
    addi $sp, $sp, 4
    jr   $ra
```

Non-Leaf Procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
  - Its return address
  - Any arguments and temporaries needed after the call
- Restore from the stack after the call

Non-Leaf Procedure Example

- C code

```
int fact (int n)
{
  if (n < 1) return 1;
  else return n * fact(n - 1);
}
```

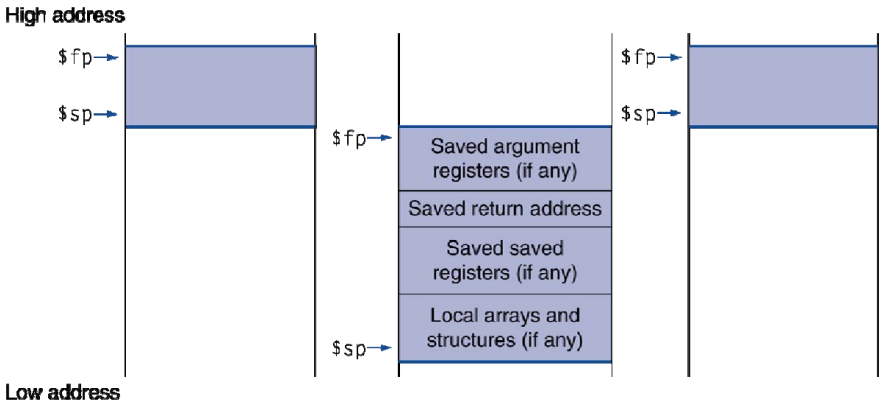
- Register usage
  - Argument n in $a0
  - Result in $v0

Non-Leaf Procedure Example

- MIPS code

```
fact:
        addi $sp, $sp, -8     # adjust stack for 2 items
        sw   $ra, 4($sp)      # save return address
        sw   $a0, 0($sp)      # save argument
        slti $t0, $a0, 1      # test for n < 1
        beq  $t0, $zero, L1
        addi $v0, $zero, 1    # if so, result is 1
        lw   $a0, 0($sp)
        lw   $ra, 4($sp)
        addi $sp, $sp, 8      #   pop 2 items from stack
        jr   $ra              #   and return
  L1:   addi $a0, $a0, -1     # else decrement n
        jal  fact             #   recursive call
        lw   $a0, 0($sp)      #   restore original n
        lw   $ra, 4($sp)      #   and return address
        addi $sp, $sp, 8      #   pop 2 items from stack
        mul  $v0, $a0, $v0    #   multiply to get result
        jr   $ra              #   and return
```

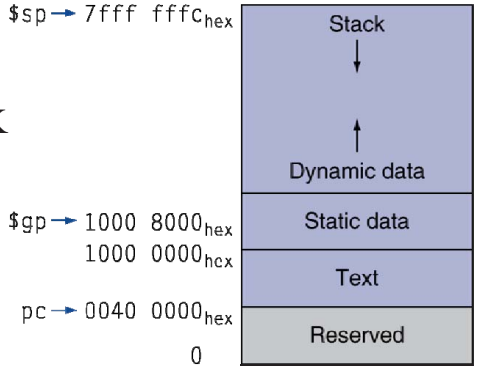

Local Data on the Stack



- Local data allocated by callee
  - Example: C automatic variables
- Procedure frame (activation record)
  - Used by some compilers to manage stack storage

Memory Layout

- Text: program code
- Static data: global variables
  - static variables in C
  - constant arrays and strings
  - $gp points to the middle of 64K block, which is combined with ±offsets for easy access
- Dynamic data: heap
  - malloc in C
  - new in Java
- Stack: local variables
  - saved registers



32-bit Immediate Operands

- 16-bit immediate constant
  - 16 bits are sufficient for most constants since they are small
- MIPS instruction to support 32-bit immediate constant

```
lui rt, constant
```

  - Copies 16-bit constant to left 16 bits of `rt`
  - Clears right 16 bits of `rt` to 0
- 32-bit immediate constant

`lhi $s0, 61`	0000 0000 0111 1101	0000 0000 0000 0000
`ori $s0, $s0, 2304`	0000 0000 0111 1101	0000 1001 0000 0000

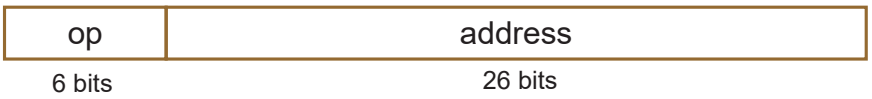
Addressing in Branches

- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward

- PC-relative addressing
  - Target address = PC + offset × 4
  - PC already incremented by 4 by this time

Addressing in Jumps

- Jump (`j` and `jal`) targets could be anywhere in text segment
  - Encode full address in instruction



- (Pseudo) Direct jump addressing
  - Target address = $PC_{31...28}$: (address × 4)

Target Addressing Example

- Loop code from earlier example
  - Assume Loop at location 80000

```
Loop: sll  $t1, $s3, 2     80000
      add  $t1, $t1, $s6   80004
      lw   $t0, 0($t1)     80008
      bne  $t0, $s5, Exit  80012
      addi $s3, $s3, 1     80016
      j    Loop            80020
Exit: …                    80024
```

0	0	19	9	4	0	
0	9	22	9	0	32	
35	9	8	0			
5	8	21	2			
8	19	19	1			
2	20000					

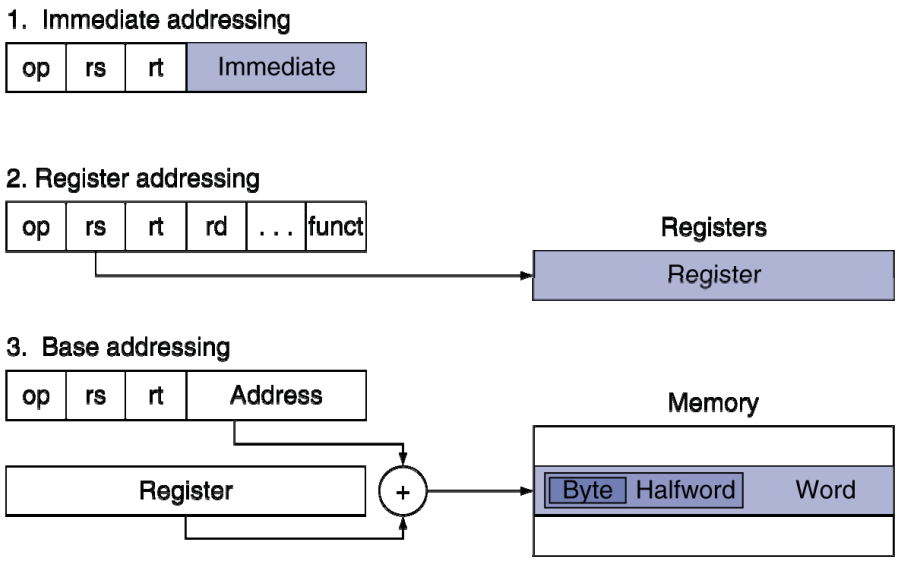
Branching Far Away

- If branch target is too far to encode with 16-bit offset, assembler rewrites the code
- Example

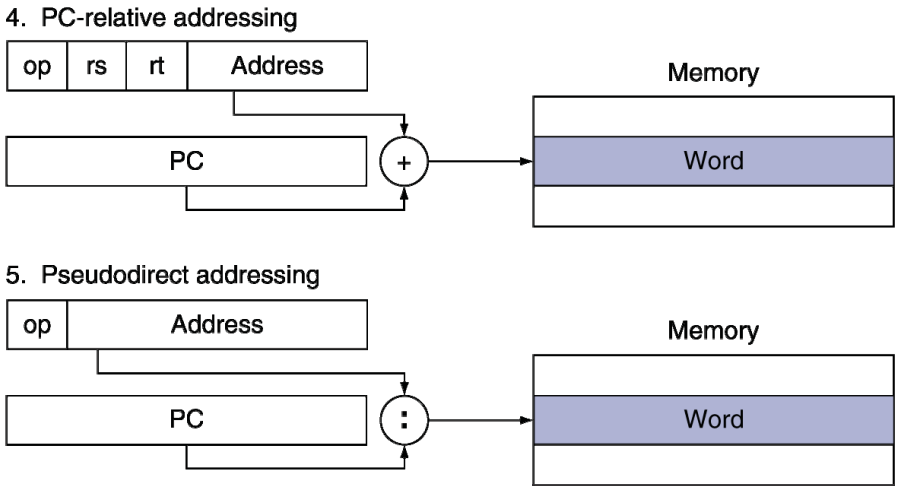
```
      beq $s0,$s1, L1
```

```
      bne $s0,$s1, L2
      j L1
L2:       …
```

Addressing Mode Summary (1)



Addressing Mode Summary (2)



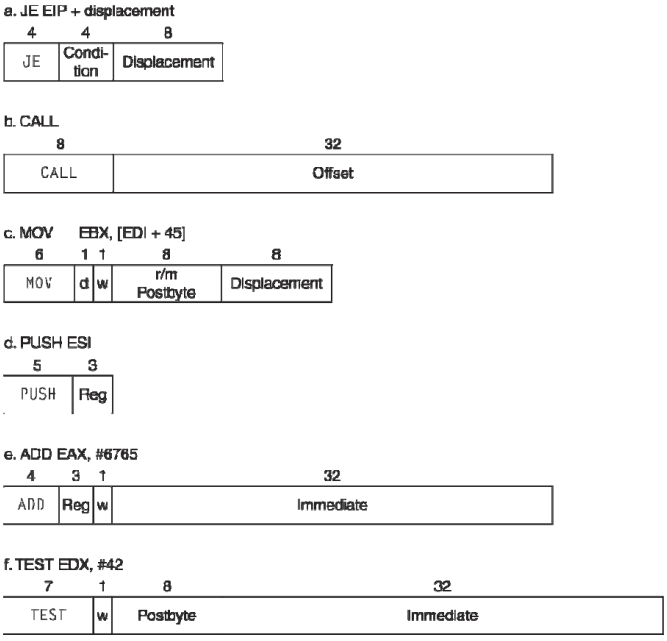
x86 Addressing Modes

- Two operands per instruction

Source/dest operand	Second source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

- Memory addressing modes
  - Address in register
  - Address = R_{base} + displacement
  - Address = $R_{base} + 2^{scale} \times R_{index}$ (scale = 0, 1, 2, or 3)
  - Address = $R_{base} + 2^{scale} \times R_{index}$ + displacement

x86 Instruction Encoding

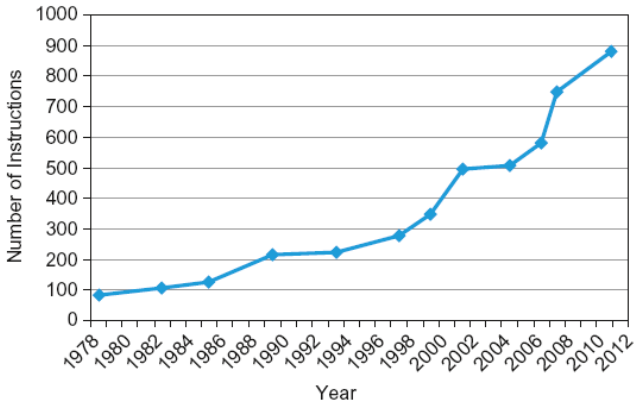


Fallacies on Instruction Set

- Powerful instruction ⇒ higher performance
  - Fewer instructions required
  - But complex instructions are hard to implement and may slow down all instructions, including simple ones
  - Compilers are good at making fast code from simple instructions
- Use assembly code for high performance
  - But modern compilers are better at dealing with modern processors
  - More lines of code ⇒ more errors and less productivity

Fallacies on Instruction Set

- Binary compatibility ⇒ instruction set doesn't change
  - Backwards binary compatibility is just okay
  - So the architecture has grown dramatically



Pitfalls

- With byte addressing, sequential ‘word addresses’ do not increment by one
  - Increment by 4 with 32 bit words
  - Even with byte addressing, word addresses are unavoidable

- Using a pointer to an automatic variable outside its defined procedure
  - To pass back a pointer to an array that is local to some procedure after leaving the procedure
  - Pointer becomes invalid when stack popped