Chapter 3
Instructions: Language of the Machine

Instructions:

- **Language of the Machine**
- **More primitive than higher level languages**
  e.g., no sophisticated control flow
- **Very restrictive**
  e.g., MIPS Arithmetic Instructions

- **We'll be working with the MIPS instruction set architecture**
  - **similar to other architectures developed since the 1980's**
  - **used by NEC, Nintendo, Silicon Graphics, Sony**

Design goals: maximize performance and minimize cost, reduce design time

3.2 MIPS arithmetic

- **All instructions have 3 operands**
- **Operand order is fixed (destination first)**

Example:

C code: `A = B + C`

MIPS code: `add $s0, $s1, $s2`

(associated with variables by compiler)

MIPS arithmetic

- **Design Principle: simplicity favors regularity. Why?**
- **Of course this complicates some things...**

C code:

```
A = B + C + D;
E = F - A;
```

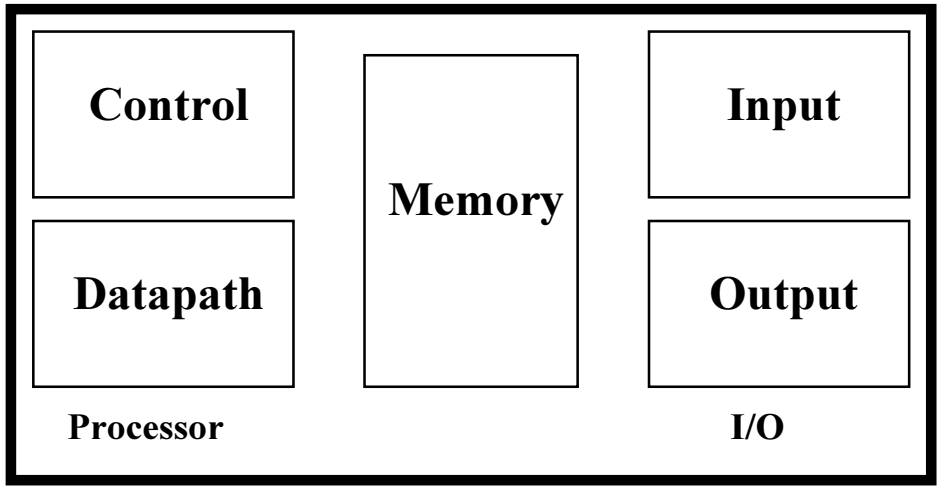
MIPS code:

```
add $t0, $s1, $s2
add $s0, $t0, $s3
sub $s4, $s5, $s0
```

- **Operands must be registers, only 32 registers provided**
- **Design Principle: smaller is faster. Why?**

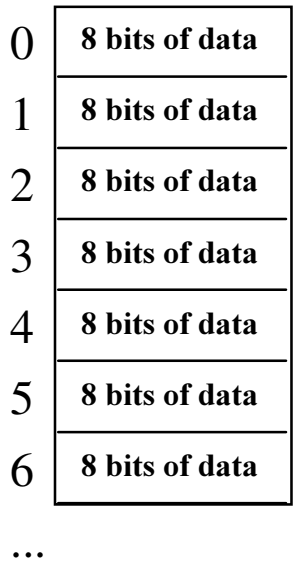
Registers vs. Memory (3.3)

- **Arithmetic instructions operands must be registers,**
  — only 32 registers provided (32 bits)
- **Compiler associates variables with registers**
- **What about programs with lots of variables**



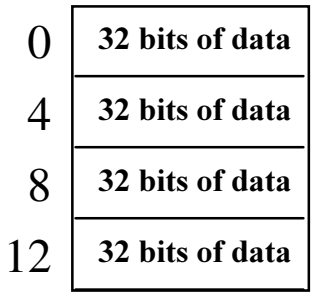
Memory Organization

- **Viewed as a large, single-dimension array, with an address.**
- **A memory address is an index into the array**
- **"Byte addressing" means that the index points to a byte of memory.**



Memory Organization

- **Bytes are nice, but most data items use larger "words"**
- **For MIPS, a word is 32 bits or 4 bytes.**



Registers hold 32 bits of data

- **2^{32} bytes with byte addresses from 0 to 2^{32}-1**
- **2^{30} words with byte addresses 0, 4, 8, ... 2^{32}-4**
- **Words are aligned**

  i.e., what are the least 2 significant bits of a word address?

Instructions

- **Load and store instructions**
- **Example:**

C code:

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

MIPS code:

```
lw $t0, 32($s3)
add $t0, $s2, $t0
sw $t0, 32($s3)
```

- **Store word has destination last**
- **Remember arithmetic operands are registers, not memory! (isto é chamado de arquitetura “load-store”)**

Exemplo com array (pag. 114)

- **Seja** `g = h + A[i];`
  onde A é um array de 100 palavras com base apontada por `$s3` **e o compilador associa as variáveis g, h e i com os registradores:**
  - `g: $s1`
  - `h: $s2`
  - `i: $s4`
- **Antes de** `$t1 ⇐ A[i]` **é necessário calcular o endereço do elemento (A+4*i):**

```
(multiply) $t1, $s4, (valor 4)
add $t1, $s3, $t1
```

- **Agora é possível ler o endereço apontado por $t1 e executar a soma**

```
lw $t0, 0($t1)        #temp $t0 ⇐ A[i]
add $s1, $s2, $t0,    # g ⇐ h + A[i]
```

Our First Example

- **Can we figure out the code?**

```
swap(int v[], int k);
{ int temp;
     temp = v[k]
     v[k] = v[k+1];
     v[k+1] = temp;
}
```

```
swap:
     muli $2, $5, 4
     add $2, $4, $2
     lw $15, 0($2)
     lw $16, 4($2)
     sw $16, 0($2)
     sw $15, 4($2)
     jr $31
```

So far we've learned:

- **MIPS**
  - **— loading words but addressing bytes**
  - **— arithmetic on registers only**

- **Instruction** **Meaning**

```
add $s1, $s2, $s3       $s1 = $s2 + $s3
sub $s1, $s2, $s3       $s1 = $s2 - $s3
lw $s1, 100($s2)        $s1 = Memory[$s2+100]
sw $s1, 100($s2)        Memory[$s2+100] = $s1
```

Machine Language (3.4)

- **Instructions, like registers and words of data, are also 32 bits long**
  - **Example:** `add $t0, $s1, $s2`
  - **registers have numbers,** `$t0=8, $s1=17, $s2=18`

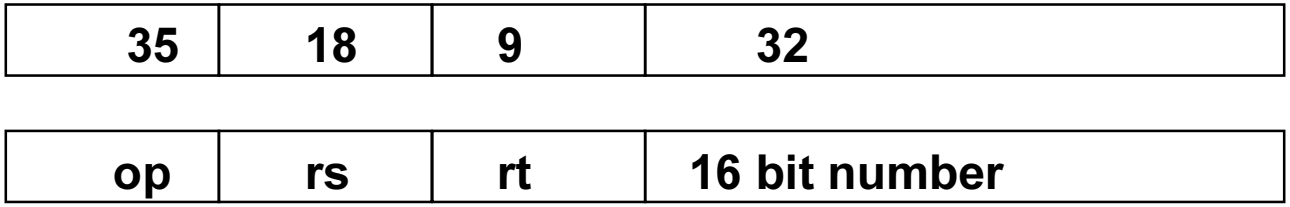
- **Instruction Format (TIPO R):**

000000	10001	10010	01000	00000	100000
op	rs	rt	rd	shamt	funct

- ***Can you guess what the field names stand for?***
  - **rs: source register**
  - **rt: target register**
  - **rd: destination register**
  - **op + funct: definem a instrução**

Machine Language

- Consider the load-word and store-word instructions,
  - What would the regularity principle have us do?
  - New principle: Good design demands a compromise
- Introduce a new type of instruction format
  - I-type for data transfer instructions
  - other format was R-type for register
- Example: `lw $t0, 32($s2)`



- Where's the compromise?

Formatos de instrução já vistos

Instrução	Formato	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32	–
sub	R	0	reg	reg	reg	0	42	–
lw	I	35	reg	reg	–	–	–	address
sw	I	45	reg	reg	–	–	–	address

Exemplo (pag 119)

C code: `A[300] = h + A[300];`

MIPS code:

```
lw $t0, 1200($t1)
add $t0, $s2, $t0
sw $t0, 1200($t1)
```

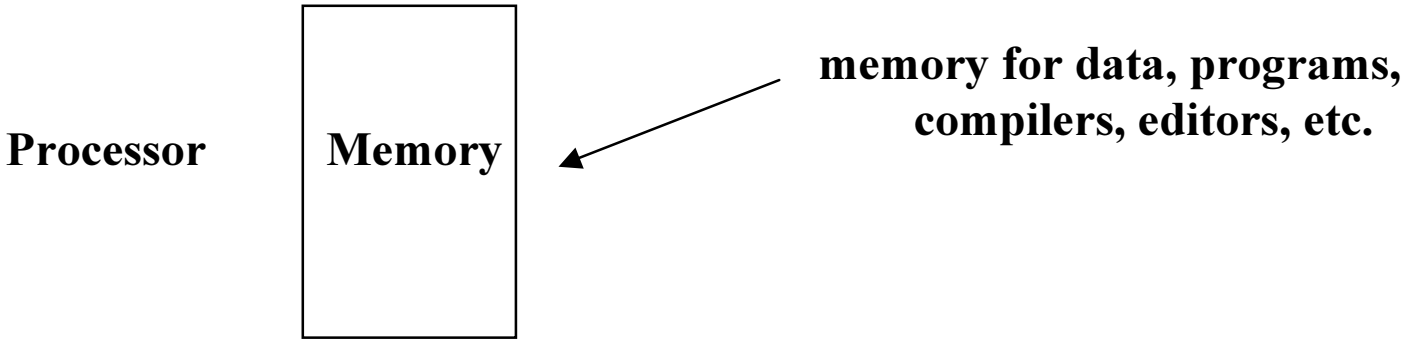
op	rs	rt	rd	address/shamt	address/funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

Instruções vistas até agora

	Form.	Exemplo						#
add	R	0	18	19	17	0	32	`add $s1, $s2, $s3`
sub	R	0	18	19	17	0	34	`sub $s1, $s2, $s3`
lw	I	35	8	17	100			`lw $s1, 100($s2)`
sw	I	43	8	17	100			`sw $s1, 100($s2)`
bits		6	5	5	5	5	6	todas com 32 bits
Form	R	op	rs	rt	rd	shamt	funct	aritmética
Form	I	op	rs	rt	address			trans. dados

Stored Program Concept

- Instructions are bits
- Programs are stored in memory
  — to be read or written just like data



- Fetch & Execute Cycle
  - Instructions are fetched and put into a special register
  - Bits in the register "control" the subsequent actions
  - Fetch the “next” instruction and continue

Control (3.5)

- **Decision making instructions**
  - **alter the control flow,**
  - **i.e., change the "next" instruction to be executed**

- **MIPS conditional branch instructions:**

```
bne $t0, $t1, Label
beq $t0, $t1, Label
```

Example (pag 123)

- **(assumir f g h i j ⇒ $s0 -> $s4)**

```
            if (i = = j) goto L1;
            f = g + h;
      L1:   f = f - i;

            beq $s3, $s4, Label    # goto label if i ≠ j
            add $s0, $s1, $s2      # faz a soma
      Label: sub $s0, $s0, s$3
```

- **e se não há label explícito no código C?**

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

assembler cria label

Control

- **MIPS unconditional branch instructions:**

```
j  label
```

- **Example:**

```
if (i!=j)                beq $s4, $s5, Lab1
    h=i+j;               add $s3, $s4, $s5
else                     j Lab2
    h=i-j;               Lab1:  sub $s3, $s4, $s5
                         Lab2:  ...
```

- ***Can you build a simple for loop?***

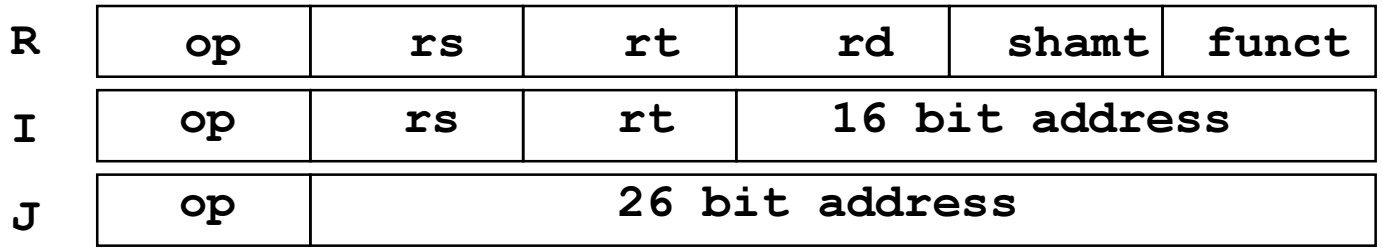
```
for (i=0; i < n; i = i + 1)

        i <- 0
label:  corpo do for
        i = i + 1
        teste (usando beq ou bne)-> label
```

So far:

- **Instruction** **Meaning**

```
add $s1,$s2,$s3   $s1 = $s2 + $s3
sub $s1,$s2,$s3   $s1 = $s2 - $s3
lw $s1,100($s2)   $s1 = Memory[$s2+100]
sw $s1,100($s2)   Memory[$s2+100] = $s1
bne $s4,$s5,L     Next instr. is at Label if $s4 ° $s5
beq $s4,$s5,L     Next instr. is at Label if $s4 = $s5
j Label           Next instr. is at Label
```

- **Formats:**



Control Flow (if then else)

```
if (i == j)          f = g + h ; else f = g - h;
    $s3  $s4         $s0 $s1 $s2
```

```
          bne $s3, $s4, Else      # goto Else if i ≠ j
          add $s0, $s1, $s2       # f = g + h
          j Exit
Else:     sub $s0, $s1, $s2       # f = g - h
Exit:     ....
```

Control Flow (slt)

- **slt (set-on-less-than):**

```
slt $t0, $s1, $s2
```

```
if  $s1 < $s2 then
   $t0 = 1
else
   $t0 = 0
```

- **Can use this instruction to build "`blt $s1, $s2, Label`"**
  — can now build general control structures
- **Note that the assembler needs a register to do this,**
  — there are policy of use conventions for registers

Policy of Use Conventions

Name	Register number	Usage
`$zero`	0	the constant value 0
`$v0-$v1`	2-3	values for results and expression evaluation
`$a0-$a3`	4-7	arguments
`$t0-$t7`	8-15	temporaries
`$s0-$s7`	16-23	saved
`$t8-$t9`	24-25	more temporaries
`$gp`	28	global pointer
`$sp`	29	stack pointer
`$fp`	30	frame pointer
`$ra`	31	return address

$1 = $at: reservado para o assembler
$26-27 = $k0-$k1: reservados para o sistema operacional

Uso do $zero

- **Zero é um valor útil**
- **Custo em hardware é (quase) nulo**
- **Útil, por exemplo, para implementar**
  - `mov $s1, $s2` **# $s1 ⇐ $s2**
  - `clear $s1` **# $s1 ⇐ 0**
  - `blt $s1, $s2, Label` **# branch on less than**

Exemplo: loop com array (pag 126)

```
Loop:   g = g + A[i];          # (g      h     i     j     A)
        i = i + j;             # ($s1    $s2   $s3   $s4   $s5)
        if (i != h) goto Loop;

Loop:   add $t1, $s3, $zero    #  $t1 ⇐ i
        multi $t1, $t1, 4      #  $t1 ⇐ i * 4  (instrução adiante)
        add $t1, $t1, $s5      # $t1 ⇐ i * 4 + A (posição do elemento)
        lw $t0, 0($t1)         # $t0 ⇐ A[i]
        add $s1, $s1, $t0      # g ⇐ g + A[i]
        add $s3, $s3, $s4      # i = i + j
        bne $s3, $s2, Loop     # goto Loop if i ≠ h
```

Exemplo: while loop (pag 127)

```
Loop:   while (save[i] = = k)   # (i      j      k      save)
        i = i + j;              # ($s3   $s4   $s5     $s6)
```

```
Loop:   add $t1, $s3, $zero     #  $t1 ⇐ i
        multi $t1, $t1, 4       #  $t1 ⇐ i * 4  (instrução adiante)
        add $t1, $t1, $s6       # $t1 ⇐ i * 4 + save
        lw $t0, 0($t1)          # $t0 ⇐ save[i]
        bne $t0, $s5, Exit      # goto Exit  if save[i] ≠  k
        add $s3, $s3, $s4       # i = i + j
        j Loop
Exit:   ....
```

(ver exercício 3.9 para otimização, reduzindo para uma instrução de desvio por ciclo)

Case / switch e jr (pag 129)

```
switch (k) {                          # (f      g      h      i      j      k)
        case 0: f=i+j; break;         # ($s0   $s1    $s2    $s3   $s4   $s5)
        case 1: f=g+h; break;         # ($t2 contém o valor 4)
        case 2: f=g-h; break;         # ($t4 contém base de JumpTable)
        case 3: f=i-j; break;
}

        (testar primeiro se k dentro de 0-3)
        add $t1, $s5, $zero     #  $t1 ⇐ k
        multi $t1, $t1, 4       #  $t1 ⇐ i * 4
        add $t1, $t1, $t4       # $t1 ⇐ k * 4 + JumpTable
        lw $t0, 0($t1)          # $t0 ⇐ endereço a ser saltado
        jr $t0                  # salta para o endereço
        ...
L0:     add $s0, $s3, $s4       # f = i + j
        j Exit                  # break
L1:     add $s0, $s1, $s2       # f = g + h
        j Exit                  # break
        ...
```

- ifs aninhados?
- quem é melhor?

Chamada de procedimentos

- **Desvio: passar parâmetros, executar (salvar contexto), retornar, (recuperar contexto)**
- **Instrução SIMPLES do MIPS ⇒ jal endereço**
  - **desvia para endereço**
  - **salva "automaticamente" endereço da próxima instrução em $ra ($31)**
- **O que acontece com jal aninhados?**
- **Solução: pilha**
  - **MIPS: implementada na memória, com $sp ($29)**
  - **ver exemplos pag 134 - 139**

push $ra

```
addi   $sp, $sp, -4
sw     $ra, 0($sp)
```

pop $ra

```
lw     $ra, 0($sp)
addi   $sp, $sp, 4
```

Chamada de procedimentos

- **contexto:**
  - **endereço de retorno**
  - **registradores tipo $sn (devem ser salvos, não podem ser alterados)**
- **salvamento do contexto**
  - **pelo “caller”**
  - **pelo “callee”**
- **argumentos passados entre o chamador e o chamado**
  - **usar os registradores $a0 = $a3**
- **temporários não precisam ser salvos**

Constants

- **Small constants are used quite frequently (50% of operands)**

  e.g., A = A + 5;
  B = B + 1;
  C = C - 18;

- **Solutions? Why not?**
  - **put 'typical constants' in memory and load them.**
  - **create hard-wired registers (like $zero) for constants like one.**

- **MIPS Instructions:**

```
addi $29, $29, 4
slti $8, $18, 10
andi $29, $29, 6
ori $29, $29, 4
```

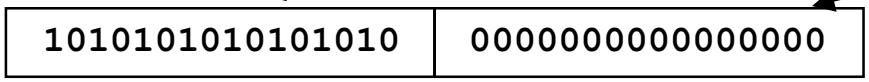
- **How do we make this work?**

How about larger constants?

- We'd like to be able to load a 32 bit constant into a register
- Must use two instructions, new "load upper immediate" instruction

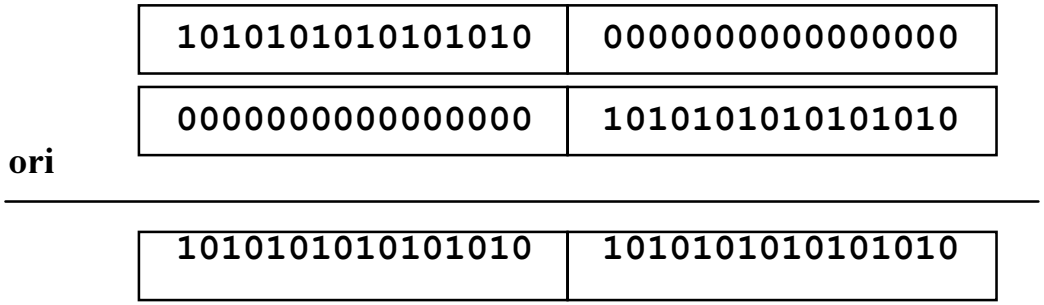
```
lui $t0, 1010101010101010
```

filled with zeros



- Then must get the lower order bits right, i.e.,

```
ori $t0, $t0, 1010101010101010
```



Assembly Language vs. Machine Language

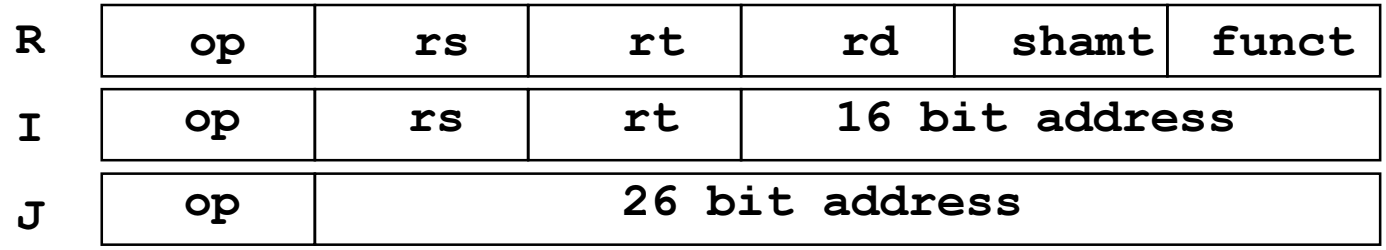
- Assembly provides convenient symbolic representation
  - much easier than writing down numbers
  - e.g., destination first
- Machine language is the underlying reality
  - e.g., destination is no longer first
- Assembly can provide 'pseudoinstructions'
  - e.g., “move $t0, $t1” exists only in Assembly
  - would be implemented using “add $t0,$t1,$zero”
- When considering performance you should count real instructions

Other Issues

- **Things we are not going to cover**
  - **linkers, loaders, memory layout**
  - **stacks, frames, recursion**
  - **manipulating strings and pointers**
  - **interrupts and exceptions**
  - **system calls and conventions**
- **Some of these we'll talk about later**
- **We've focused on architectural issues**
  - **basics of MIPS assembly language and machine code**
  - **we'll build a processor to execute these instructions.**

Overview of MIPS

- simple instructions all 32 bits wide
- very structured, no unnecessary baggage
- only three instruction formats



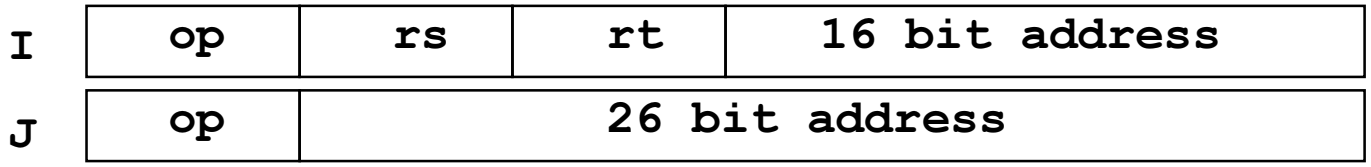
- rely on compiler to achieve performance
  — what are the compiler's goals?
- help compiler where we can

Addresses in Branches and Jumps

- **Instructions:**

`bne $t4,$t5,Label`	Next instruction is at Label if `$t4` ≠ `$t5`
`beq $t4,$t5,Label`	Next instruction is at Label if `$t4 = $t5`
`j Label`	Next instruction is at Label

- **Formats:**



- **Addresses are not 32 bits**
  - **— How do we handle this with load and store instructions?**

Addresses in Branches

- **Instructions:**

  `bne $t4,$t5,Label` **Next instruction is at Label if $t4 ≠ $t5**

  `beq $t4,$t5,Label` **Next instruction is at Label if $t4 = $t5**

- **Formats:**

- **Could specify a register (like lw and sw) and add it to address**
  - **use Instruction Address Register (PC = program counter)**
  - **most branches are local (principle of locality)**
- **Jump instructions just use high order bits of PC**
  - **address boundaries of 256 MB**

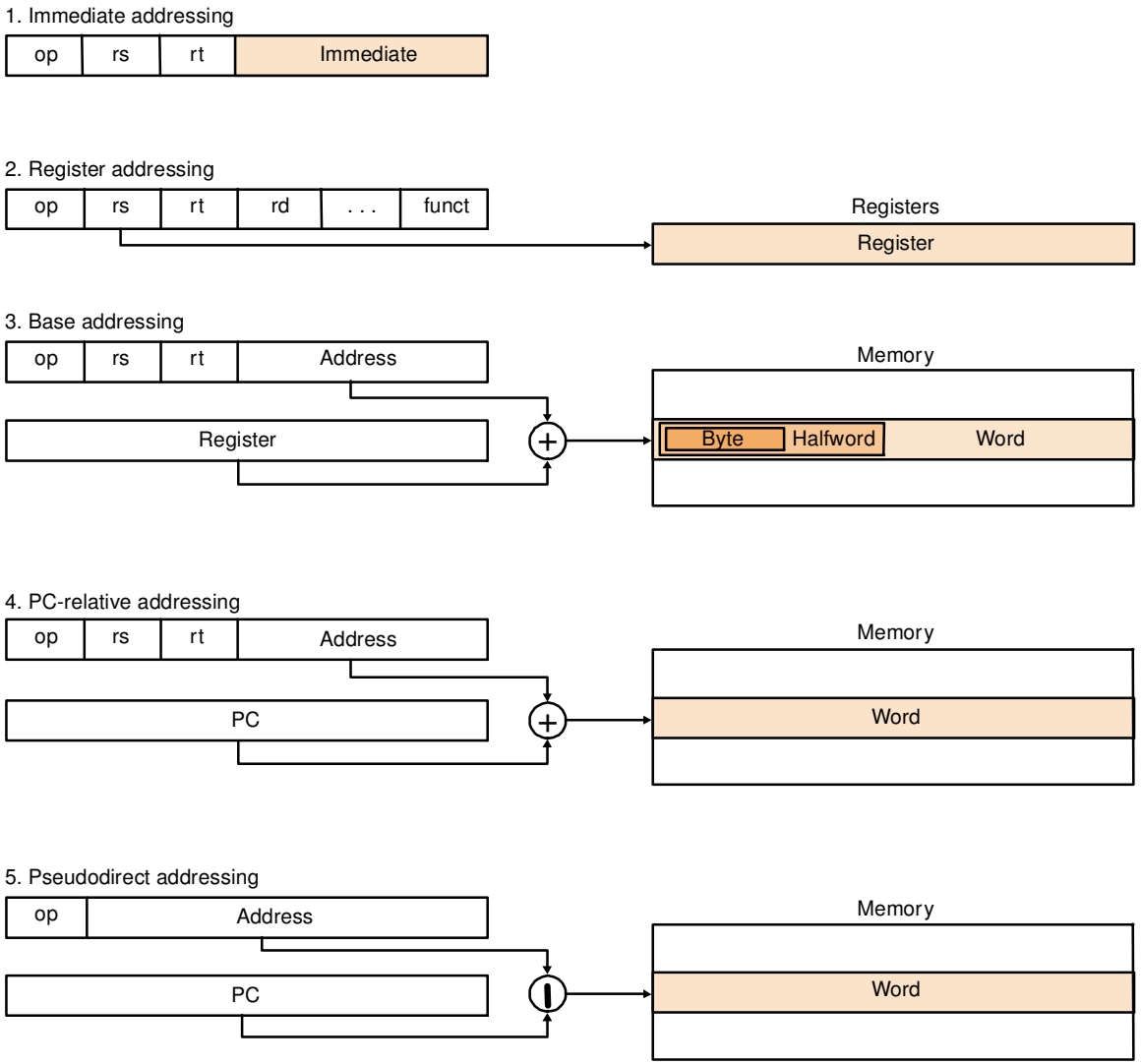
To summarize:

MIPS operands

Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`	Three operands; data in registers
	subtract	`sub $s1, $s2, $s3`	`$s1 = $s2 - $s3`	Three operands; data in registers
	add immediate	`addi $s1, $s2, 100`	`$s1 = $s2 + 100`	Used to add constants
Data transfer	load word	`lw $s1, 100($s2)`	`$s1` = Memory[`$s2` + 100]	Word from memory to register
	store word	`sw $s1, 100($s2)`	Memory[`$s2` + 100] = $s1	Word from register to memory
	load byte	`lb $s1, 100($s2)`	`$s1` = Memory[`$s2` + 100]	Byte from memory to register
	store byte	`sb $s1, 100($s2)`	Memory[`$s2` + 100] = $s1	Byte from register to memory
	load upper immediate	`lui $s1, 100`	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
Conditional branch	branch on equal	`beq $s1, $s2, 25`	if (`$s1 == $s2`) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1, $s2, 25`	if (`$s1 != $s2`) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1, $s2, $s3`	if (`$s2 < $s3`) `$s1` = 1; else `$s1` = 0	Compare less than; for beq, bne
	set less than immediate	`slti $s1, $s2, 100`	if (`$s2 < 100`) `$s1` = 1; else `$s1` = 0	Compare less than constant
Uncondi-tional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to `$ra`	For switch, procedure return
	jump and link	`jal 2500`	`$ra` = PC + 4; go to 10000	For procedure call



Alternative Architectures

- **Design alternative:**
  - **provide more powerful operations**
  - **goal is to reduce number of instructions executed**
  - **danger is a slower cycle time and/or a higher CPI**
- **Sometimes referred to as "RISC vs. CISC"**
  - **virtually all new instruction sets since 1982 have been RISC**
  - **VAX: minimize code size, make assembly language easy**

    instructions from 1 to 54 bytes long!
- **We'll look at PowerPC and 80x86**

Um exemplo completo

- **mostrar transparência do sort**
  - **para ilustrar estrutura geral**
  - **não será cobrado**
- **array e pointer**
  - **não será cobrado**

PowerPC (Motorola, Apple, IBM)

- 32 registradores de 32 bits, instruções de 32 bits
- Indexed addressing
  - example: `lw $t1,$a0+$s3  #$t1=Memory[$a0+$s3]`
  - What do we have to do in MIPS?

- Update addressing
  - update a register as part of load (for marching through arrays)
  - example: `lwu $t0,4($s3) #$t0=Memory[$s3+4];$s3=$s3+4`
  - What do we have to do in MIPS?
- Others:
  - load multiple/store multiple
  - a special counter register "`bc Loop`"

decrement counter, if not 0 goto loop

80x86

- 1978: The Intel 8086 is announced (16 bit architecture)
- 1980: The 8087 floating point coprocessor is added
- 1982: The 80286 increases address space to 24 bits, +instructions
- 1985: The 80386 extends to 32 bits, new addressing modes
- 1989-1995: The 80486, Pentium, Pentium Pro add a few instructions (mostly designed for higher performance)
- 1997: MMX is added

"This history illustrates the impact of the "golden handcuffs" of compatibility

"adding new features as someone might add clothing to a packed bag"

"an architecture that is difficult to explain and impossible to love"

A dominant architecture: 80x86

- See your textbook for a more detailed description
- Complexity:
  - Instructions from 1 to 17 bytes long
  - one operand must act as both a source and destination
  - one operand can come from memory
  - complex addressing modes
    e.g., “base or scaled index with 8 or 32 bit displacement”
- Saving grace:
  - the most frequently used instructions are not too difficult to build
  - compilers avoid the portions of the architecture that are slow

“what the 80x86 lacks in style is made up in quantity,
making it beautiful from the right perspective”

Conclusão

- **Erro: instruções mais poderosas aumentam desempenho**
- **VAX:**
  - **CALL: salva endereço de retorno, nº de parâmetros, quaisquer registros modificados e valor antigo do SP**
  - **instrução para apagar lista duplamente ligada**
- **IBM 360:**
  - **10 instruções mais freqüentes: 80% das ocorrências**
  - **16 instruções mais freqüentes: 90% das ocorrências**
  - **21 instruções mais freqüentes: 95% das ocorrências**
  - **30 instruções mais freqüentes: 99% das ocorrências**
- **MIPS**

classe	instr	gcc	spice
arit.	`add, sub, addi`	48%	50%
transf. dados	`lw, sw, lb, sb, lui`	33%	41%
desvio cond.	`beq, bne, slt, slti`	17%	8%
jump	`j, jr, jal`	2%	1%

Summary

- Instruction complexity is only one variable
  - lower instruction count vs. higher CPI / lower clock rate
- Design Principles:
  - simplicity favors regularity (facilidade de projeto)
  - smaller is faster
  - good design demands compromise
  - make the common case fast (RISC)
- Instruction set architecture
  - a very important abstraction indeed!

Máquinas de 0, 1, 2 e 3 endereços

X = A * B + C * C onde X, A, B, C são endereços de posições de memória

Um endereço

```
LOAD A
MULTIPLY B
STORE T
LOAD C
MULTIPLY C
ADD T
STORE X
```

AC ⇐ AC op END

Dois endereços

```
MOVE T, A
MULTIPLY T, B
MOVE X, C
MULTIPLY X, C
ADD X, T
```

END1 ⇐ END1 op END2

Três endereços

```
MULTIPLY T, A, B
MULTIPLY X, C, C
ADD X, X, T
```

END1 ⇐ END2 op END3

Load-Store

```
load R1, A
load R2, B
load R3, C
mult R1, R1, R2
mult R3, R3, R3
add R1, R1, R3
store R1, X
```

R1 ⇐ R2 op R3

Zero endereços

```
push A
push B
mult
push C
push C
mult
add
pop X
```

tos ⇐ tos op tos

Máquinas de 0, 1, 2 e 3 endereços

- **Qual é o melhor?**
  - **tamanho do código fonte**
  - **tamanho do código objeto**
  - **tempo de execução**
  - **simplicidade e desempenho do hardware para suportar arquitetura**