Chapter 3

©1998 Morgan Kaufmann Publishers

MIPS Instructions

- Instruction / Meaning

```
Instruction          Meaning

add $s1,$s2,$s3      $s1 = $s2 + $s3
sub $s1,$s2,$s3      $s1 = $s2 – $s3

addi $s1,$s2,4       $s1 = $s2 + 4
ori  $s1,$s2,4       $s2 = $s2 | 4

lw $s1,100($s2)      $s1 = Memory[$s2+100]
sw $s1,100($s2)      Memory[$s2+100] = $s1

bne $s4,$s5,Label    Next instr. is at Label if $s4 ¹ $s5
beq $s4,$s5,Label    Next instr. is at Label if $s4 = $s5
slt $t1,$s2,$s3       if $s2 < $s3, $t1 = 1 else $t1 = 0

j Label              Next instr. is at Label
jr $s1               Next instr is in register $s1
jal Label            Jump and link procedure at Label
```


©1998 Morgan Kaufmann Publishers

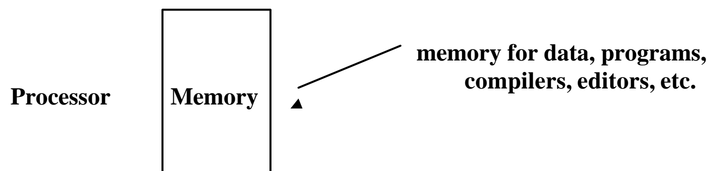
Assembly Language vs. Machine Language

- 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”

©1998 Morgan Kaufmann Publishers

Stored Program Concept

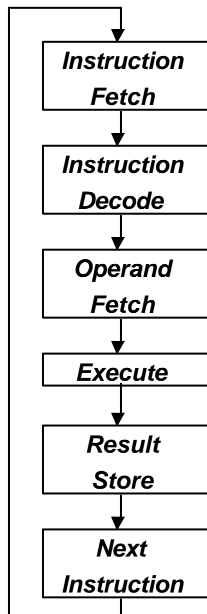
- Instructions are bits
- Programs are stored in memory
  — to be read or written just like data



- 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

©1998 Morgan Kaufmann Publishers

Instruction Set Architecture: What Must be Specified?



° Instruction Format or Encoding
  – how is it decoded?
° Location of operands and result
  – where other than memory?
  – how many explicit operands?
  – how are memory operands located?
  – which can or cannot be in memory?
° Data type and Size
° Operations
  – what are supported
° Successor instruction
  – jumps, conditions, branches

- fetch-decode-execute is implicit!

©1998 Morgan Kaufmann Publishers

MIPS Design Principles

- Reduced Instruction Set Computers (RISC) design philosophy
- Principles guiding Instruction Set Design
  - Smaller is faster
    - Example: Only 32 registers in MIPS
  - Simplicity favors regularity
  - Good design demands compromise
  - Make the common case fast

©1998 Morgan Kaufmann Publishers

Machine Language: R Format

- Instructions, like registers and words of data, are also 32 bits long
  - Example: `add $t0, $s1, $s2`
  - registers have numbers, `$t0=9, $s1=17, $s2=18`
- Instruction Format:

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

- R Format

7

©1998 Morgan Kaufmann Publishers

Machine Language: I format

- 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)`

35	18	9	32
op	rs	rt	16 bit number

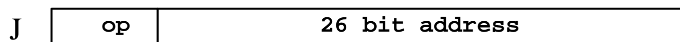
- Where's the compromise?

8

©1998 Morgan Kaufmann Publishers

Machine Language: J Format

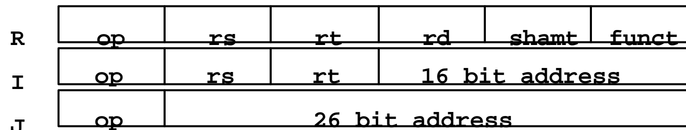
- Jump (j) , Jump and link (jal) instructions have two fields
  - Opcode
  - Address
- Instruction should be 32 bits (Regularity principle)
  - 6 bits for opcode
  - 26 bits for address



©1998 Morgan Kaufmann Publishers

MIPS Instruction Formats

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



©1998 Morgan Kaufmann Publishers

What about other instructions

- slt $t0, $s1, $s2
  - 3 operands all registers ⇒ use R format
- beq $s1,$s2, Label
  - 2 registers + address ⇒ use I format
- addi $s1,$s1, 4
  - 2 registers + immediate value ⇒ use I format
- jr $t1
  - 1 register
  - R format

©1998 Morgan Kaufmann Publishers

Implications of design choices

- Using I format for arithmetic instructions with immediate operands
  - Only 16 bits for immediate field
  - Constants have to fit in 16 bits
- Using I format for branch instructions
  - Only 16 bits in immediate field
  - But 32 bits needed for branch address
- J format
  - Only 26 bits for address field
  - But 32 bits needed for Jump address

©1998 Morgan Kaufmann Publishers

Constants

- Small constants are used quite frequently (50% of operands)
  e.g., A = A + 5;
  B = B + 1;
  C = C - 18;
- So in most programs, constants will fit in 16 bits allocated for immediate field
- Design Principle: Make the common case fast
  - Common case: constant is small
  - Only need to use one instruction in the common case

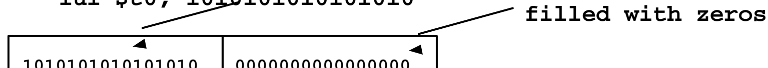
©1998 Morgan Kaufmann Publishers

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**

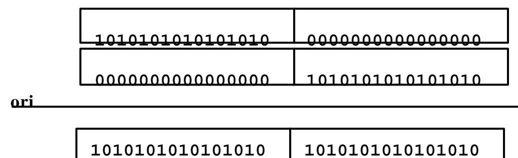
```
lui $t0, 1010101010101010
```

filled with zeros



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

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



1010101010101010 | 1010101010101010

©1998 Morgan Kaufmann Publishers

Addresses in Branches and Jumps

- **Instructions:**

`bne $t4,$t5,Label`	Next instruction is at Label if `$t4 ¹ $t5`
`beq $t4,$t5,Label`	Next instruction is at Label if `$t4 = $t5`
`j Label`	Next instruction is at Label

- **Formats:**

I	op	rs	rt	16 bit address
J	op	26 bit address		

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

©1998 Morgan Kaufmann Publishers

Addresses in Branches

- **Instructions:**

bne $t4,$t5,Label	**Next instruction is at Label if $t4¹ $t5**
beq $t4,$t5,Label	**Next instruction is at Label if $t4=$t5**

- **Formats:**

I	op	rs	rt	16 bit address

- **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**

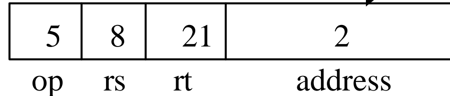
©1998 Morgan Kaufmann Publishers

Addressing in branches

- Immediate field is 16 bits but we need an address that is 32 bits
- Obtain address using *PC-relative addressing*
  - On branch, new PC = PC + immediate field in branch instruction
  - Actually, new PC = (PC+4) + immediate field in branch instruction

```
80000   Loop:  mult  $9, $19, $10
80004          lw    $8, Sstart($9)
80008          bne   $8, $21, Exit
80012          add   $19,$19,$20
80016          j  Loop
80020   Exit:
```

Immediate field contains the distance in words between PC+4 and branch target address



op rs rt address

I format

©1998 Morgan Kaufmann Publishers

Uncommon Case for branches

- beq $18, $19, L1

  replaced by

```
        bne  $18, $19, L2
        j   L1
   L2:
```

Make the common case fast

one instruction for most branches

©1998 Morgan Kaufmann Publishers

Addressing in Jumps

- J format has 26 bits in address field
  - How to get 32 bits?
- Assume that jump address is a word address
- 26 + 2 (least significant bits) = 28
- Get 4 most significant bits from PC
  - 4 + 26 + 2 = 32
  - Implication: can only jump within a 2^{28} = 256 MB block of addresses
  - Loader and linker must be careful to avoid placing a program across an address boundary of 256 MB

©1998 Morgan Kaufmann Publishers

To summarize:

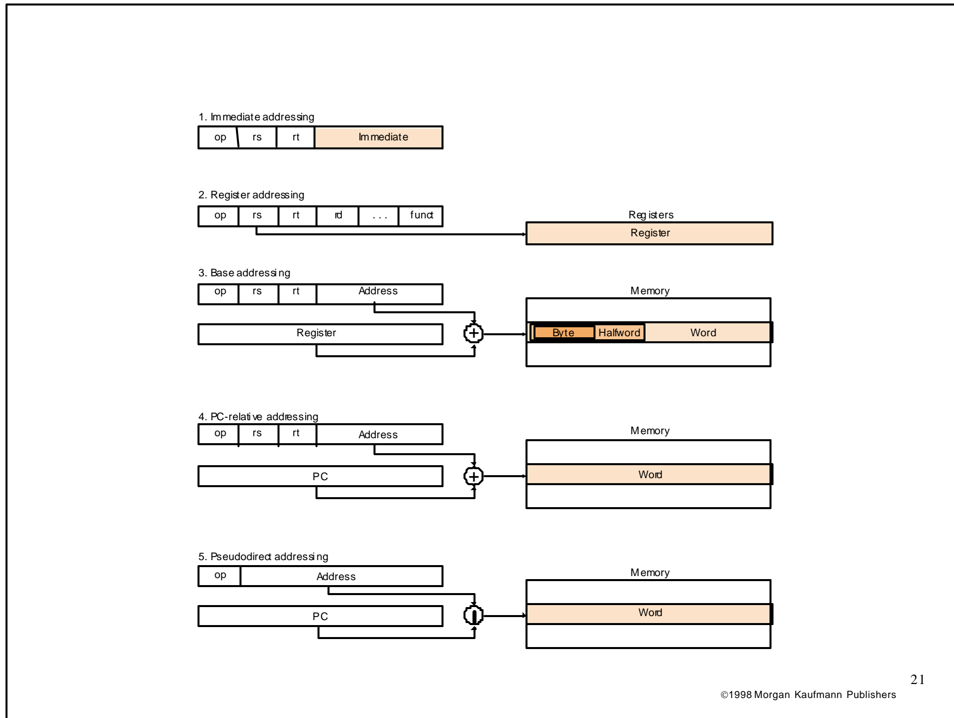
MIPS operands

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

©1998 Morgan Kaufmann Publishers



©1998 Morgan Kaufmann Publishers

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**

©1998 Morgan Kaufmann Publishers

PowerPC

- 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

©1998 Morgan Kaufmann Publishers

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"

©1998 Morgan Kaufmann Publishers

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”*

©1998 Morgan Kaufmann Publishers

Summary

- Design Principles:
  - simplicity favors regularity
  - smaller is faster
  - good design demands compromise
  - make the common case fast

©1998 Morgan Kaufmann Publishers