

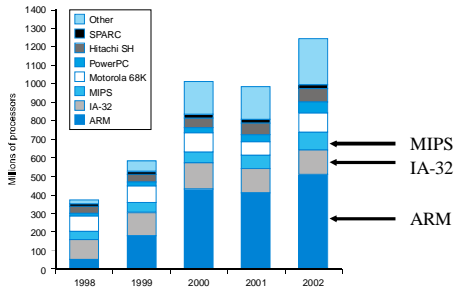
## Chapter 2

## Instruction Set Architecture:

- **Instruction: Language of the Machine**
- **We'll be working with the MIPS instruction set architecture**
  - Almost 100 million MIPS processors manufactured in 2002
  - Used in real systems: NEC, Nintendo, Cisco, Silicon Graphics, Sony, ...
    - Various routers from Cisco
    - Game machines like the Sony Playstation 2

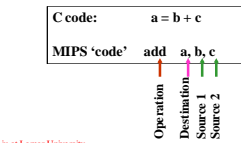


## Instruction Set Architecture:



## Operations of Computer Hardware

- **Design principle: Keep hardware simple; Simplicity favors regularity**
- **MIPS arithmetic instruction**
  - Perform *only* one operation ← Simplicity
    - add, sub, mul, div
  - Must always have *exactly* three operands ← Regularity
    - One destination and two sources
    - Operand order is fixed
      - Destination first



## Operations of Computer Hardware

- Of course this complicates some things...

C code:  $a = b + c + d;$

MIPS code: add a, b, c  
add a, a, d

## Operands of Computer Hardware

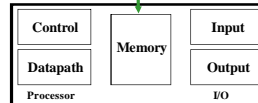
- **Design Principle: smaller is faster**
- **MIPS arithmetic instruction**
  - Operands must be registers
  - Limited number of registers
    - Different from programming languages
    - Only 32 registers provided
    - Why?
      - Large # of registers may increase clock cycle time because it takes electronic signals longer when they must travel further
    - Balance
      - Designer: Keep the clock cycle fast
      - Programmer: More register
  - Each register contains 32 bits
    - Why? - Later

## Operands of Computer Hardware

- Register names
  - MIPS convention
    - Two-character names following a \$
      - \$s0, \$s1, \$t0, \$t1, ...
  - Use numbers
    - 0 .. 31
- Use
  - \$zero: Constant value 0
  - \$k0-\$k1: Reserved for OS kernel
  - \$gp, \$sp, \$fp, \$ra: Special use
  - \$s0-\$s7: Saved temporaries
  - \$a0-\$a3: Arguments
  - \$t0-\$t7: Temporaries

## Registers vs. Memory

- Arithmetic instructions operands must be registers
  - Only limited registers provided
    - MIPS: 32 registers
  - Data width is limited
    - MIPS: 32-bit register
- What about programs with lots of variables?

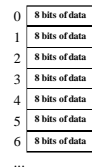


## Registers vs. Memory

- Many programs
  - More variables than registers in computers
  - Compiler associates variables with registers
    - MIPS
      - Keep most frequently used variables in registers
      - Loads and stores move variables from registers and memory
      - Spilling registers
        - Process of putting less frequently used variables into memory
  - Data is more useful when in a register
    - MIPS arithmetic instruction
      - Read two registers, operate on them, and write result
      - Data transfer instruction only read/write one operand
- Compilers must use registers efficiently to achieve highest performance

## 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



An array of bytes begins at address 1000

- The first array element is at address 1000
- The second element is at address 1001

For example

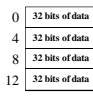
\$a0 contains 1000

lw \$t0, 0(\$a0) accesses the first byte of the array

lw \$t0, 5(\$a0) access the sixth byte of the array, at address 1005

## Memory Organization

- Bytes are nice
  - 8-bit bytes are useful
  - Most architectures address individual bytes
- However, most data items use larger "words"
- For MIPS, a word is 32 bits or 4 bytes
  - Alignment restriction
    - Words must start at addresses that are multiples of 4



• An array of bytes begins at address 1000

- The first array element is at address 1000
- The second element is at address 1004

For example

\$a0 contains 1000

lw \$t0, 0(\$a0) accesses the first word of the array 1000

lw \$t0, 12(\$a0) access the fourth word of the array at address 1012

## Memory Organization

- $2^{32}$  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[12] = h + A[8];`

MIPS code: `lw $t0, 32($s3)  
add $t0, $s2, $t0  
sw $t0, 48($s3)`

- Remember arithmetic operands are registers, not memory!

**Can't write:** `add 48($s3), $s2, 32($s3)`

## Operands of Computer Hardware

- How do we get constant data into registers?
  - Use data transfer instructions to load from memory
    - For example
      - `lw $t1, 4($t0) # $t1=mem[4+$t0]`
    - Constant data occur more frequently
      - Loading from memory is too slow
  - **Design principle: Make the common case faster**
    - Use immediate value
      - Second source operand
      - For example
        - `addi $t1, $t0, 10 # $t1=$t0+10`
    - Use \$zero and immediate value
      - For example
        - `addi $t1, $0, 10 # $t1=$0+10`

## Operands of Computer Hardware

- Why doesn't MIPS have a subtract immediate instruction?
  - Negative constant appear much less frequently
    - Java and C
  - Immediate field
    - Positive constant
    - Negative constant
  - add with negative constant
    - Equal to subtract immediate

Chapter 2 - 2

## Our First Example

- Can you 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  
  ??
```

## Examples:

```
.data  
array1: .space 12 # declare 12 bytes of storage to hold array of 3 integers  
        .text  
__start: la $t0, array1 # load base address of array into register $t0  
        li $t1, 5 # $t1 = 5 ("load immediate")  
        sw $t1, ($t0) # first array element set to 5; indirect addressing  
        li $t1, 13 # $t1 = 13  
        sw $t1, 4($t0) # second array element set to 13  
        li $t1, -7 # $t1 = -7  
        sw $t1, 8($t0) # third array element set to -7  
        done
```

```
li $v0, 4 # load appropriate system call code into register $v0;  
# code for printing string is 4  
la $a0, string1 # load address of string to be printed into $a0  
syscall # call operating system to perform print operation
```

## Reference

- MIPS instruction set and SPIM
  - Tutorial
  - Examples

## So far we've learned:

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

```

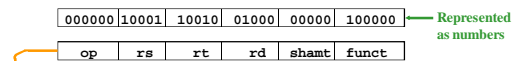
add $s1, $s2, $s3    $s1 = $s2 + $s3
sub $s1, $s2, $s3    $s1 = $s2 - $s3
lw  $s1, 100($s2)    $s1 = Memory[$s2+100]
sw  $s1, 100($s2)    Memory[$s2+100] = $s1

```

## Machine Language

- Instructions, like registers and words of data, are also 32 bits long
  - Example: add \$t1, \$s1, \$s2
  - registers have numbers, \$t1=9, \$s1=17, \$s2=18

- Instruction Format:



- Can you guess what the field names stand for?

## Machine Language

- **Design principle: Good design demands good compromises**
- All instructions have the same length
  - There are different instructions?
    - Different instruction formats for different instructions
- Instruction formats
  - R-Type
    - Arithmetic instructions
    - For register
  - I-Type
    - Immediate and data transfer instructions
  - J-Type
    - Jump instruction

## Machine Language

- **Design**
  - Multiple formats complicate hardware
  - Solution
    - Keep format similar to reduce complexity

- Instruction Formats:

R-Format    

op	rs	rt	rd	shamt	funct
----	----	----	----	-------	-------

I-Format    

op	rs	rt	address/constant
----	----	----	------------------

J-Format    

op	target address
----	----------------

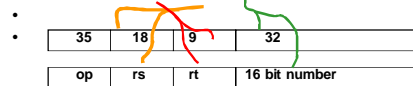
Field width (in bit)    

6	5	5	5	6	5
---	---	---	---	---	---

## Machine Language

- I-type for data transfer instructions

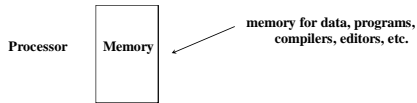
- Example: lw \$t0, 32(\$s2)



- Example: lw \$t0, 32(\$s2)

## Stored Program Concept

- **Instruction**
  - Represented as numbers
  - Program are stored in memory to be read and write, just like numbers



- **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

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

```
bne $t0, $t1, Label # branch on not equal
beq $t0, $t1, Label # branch on equal
```

- **Example:**    if (i==j) h = i + j;

```
      bne $s0, $s1, Label
      add $s3, $s0, $s1
Label: ....
```

## Control

- **Decisions are important**
  - Choose between two alternatives
    - if statement
  - Iterate a computation
    - loop

```
for (i=100; i<200; i++)
A[i]=A[i]+C;
          Loop: lw $t0, 0($t1) # $t0=A[i]
              add $t3, $t0, $t2 # $t3=$t0+$t2
                      # $t3=A[i]+C
              sw $t3, 0($t1) # A[i]=$t3=A[i]+C
              addi $t1, $t1, 4 # $t1=$t1+4
                      # i++
              bne $t1, $t4, Loop # $t1=?=$t4
```

## Control

- **MIPS unconditional branch instructions:**

```
j label # jump
```
- **Example:**

```
if (i1=j)                beq $s4, $s5, Lab1
          h=i+j;        add $s3, $s4, $s5
else                    j Lab2
          h=i-j;        Lab1: sub $s3, $s4, $s5
                          Lab2: ...
```

## Control

- **Jump address table**
  - Table of addresses of alternative instruction sequences
  - **Jump register instruction, jr**
    - Unconditional jump to address specified in a register
    - Example: jr \$31

## 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,Label Next instr. is at Label if $s4 ≠ $s5
beq $s4,$s5,Label Next instr. is at Label if $s4 = $s5
j Label             Next instr. is at Label
```

- **Formats:**

|   |    |    |    |                |       |       |
|---|----|----|----|----------------|-------|-------|
| R | op | rs | rt | rd             | shamt | funct |
| I | op | rs | rt | 16 bit address |       |       |
| J | op |    |    | 26 bit address |       |       |

## Control Flow

- Pseudoinstruction
  - Appear in assembly language
  - Not implemented in hardware
  - Assembler
    - Accept these type of instructions
    - Translate instructions into machine language
- Example
  - bit
  - move

```
Move $t0, $t1    # $t0=$t1
=====
Add $t0, $zero, $t1 # $t0=$0+$t1
```

## Control Flow

- We have: beq, bne, what about Branch-if-less-than?
- MIPS instruction:
 

```
if $s1 < $s2 then
    $t0 = 1
else
    $t0 = 0
```
- Can use this instruction to build "bit \$s1, \$s2, Label"
  - can now build general control structures

```
blt $t0, $t1, Label # branch if less than
=====
slt $sat, $t0, $t1 # $sat = 1 if $t0 < $t1
bne $sat, $0, Label # branch if $sat != 0
```
- Note that the assembler needs a register to do this,
  - there are policy of use conventions for registers
    - \$at-Assembler temporary
    - Reserved for assembler

## Constants

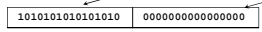
- Small constants are used quite frequently (50% of operands)
  - e.g., A = A + 5;
  - B = B + 1;
  - C = C - 18;
- Solutions? Why not?
  - put 'typical constants' in memory and load them.
  - create hard-wired registers (like \$zero) for constants like one.
- MIPS Instructions:
 

```
addi $29, $29, 4
slli $8, $18, 10
andi $29, $29, 6
ori $29, $29, 4
```
- Design Principle: Make the common case fast. Which format?

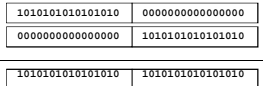
## How about larger constants?

- 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
```


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

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



## Chapter 2 - 3

## Overview of MIPS

- Simple instructions all 32 bits wide
- Very structured, no unnecessary baggage
- Instruction formats
  - R-Type
    - Arithmetic instructions
    - For register
  - I-Type
    - Immediate and data transfer instructions
  - J-Type
    - Jump instruction
- Only three instruction formats

|   |                |    |    |                |       |       |
|---|----------------|----|----|----------------|-------|-------|
| R | op             | rs | rt | rd             | shamt | funct |
| I | op             | rs | rt | 16 bit address |       |       |
| J | 26 bit address |    |    |                |       |       |

- 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 not equal to \$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 |    |                |

*No program could be bigger than 2<sup>16</sup>?  
Far too small to be a realistic option!*
- Addresses are not 32 bits
  - How do we handle this with load and store instructions?

## Addresses in Branches

- Formats:
 

|   |    |    |    |                |
|---|----|----|----|----------------|
| I | op | rs | rt | 16 bit address |
|---|----|----|----|----------------|
- Program counter (PC) or instruction address register
  - A register hold the address of current instruction being executed
- Conditional branches
  - Found in loops and if statements
  - Tend to branch to a nearby instruction (**Principle of locality**)
    - For example
      - For SPEC CPU 2000 benchmarks, half of all condition branches go to locations less than 16 instructions away
    - Solution
      - Which register should be used?
        - PC = register + Branch address
        - PC-relative addressing: PC = PC + Branch address
          - Allow branch +/- 2<sup>15</sup> words of the current instruction (instruction at PC+4)
        - Addressing refer to # of words to the next instruction instead of # of bytes

## Addresses in Branches

- Formats:
 

|   |    |    |    |                |
|---|----|----|----|----------------|
| I | op | rs | rt | 16 bit address |
|---|----|----|----|----------------|
- Conditional branches
  - Occasionally, branch far away
    - PC-relative addressing: PC = PC + Branch address
      - Not large enough
    - Solution
      - Insert an unconditional jump to offer a much greater branching distance
      - Example

```

beq $s0, $s1, L1
=====
bne $s0, $s1, L2
j    L1
L2:
    
```

*Replaced by*

## Addresses in Branches and Jumps

- Formats:
 

|   |    |                |
|---|----|----------------|
| J | op | 26-bit address |
|---|----|----------------|
- J-type instruction
  - Jump and jump-and-link instructions
    - Invoke procedures not near the call
    - 26-bit word address means 28-bit byte address
      - Where are the other 4 bits of address?
        - Leave the upper 4-bit of PC unchanged
    - Address boundary 256MB (64 million instructions)
      - Otherwise, replaced by jump register instruction preceded by other instructions to load the full 32-bit address into a register
    - Addressing refer to # of words to the next instruction instead of # of bytes

### Byte address:

```

jump  xxxxxx|000000000000000000000000
jump  xxxxxx|111111111111111111111111
    
```

```

PC    010101|000000000000000000000000
PC    010101|111111111111111111111111
    
```

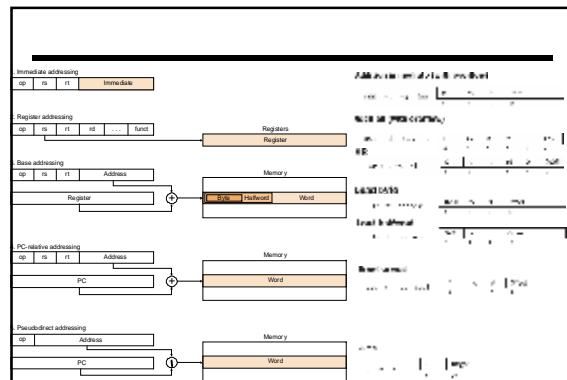
### Word address:

```

jump  xxxxxx|000000000000000000000000
jump  xxxxxx|111111111111111111111111
    
```

```

PC    0101|000000000000000000000000
PC    0101|111111111111111111111111
    
```



## PC-relative addressing

```

While (save[i] == k)
  i+=1;
loop: sll $t1, $s3, 2      # Temp reg $t1=4*i
      add $t1, $t1, $s6    # $t1 = addr. of save[i]; $s6 base addr.
      lw $t0, 0($t1)      # Temp reg $t0 = save[i]
      bne $t0, $s5, Exit  # go to Exit if save[i] not equal to k
      addi $s3, $s3, 1    # i = i+1
      j loop              # go to loop
Exit:
  
```

| Loop: sll | 80000 | 0      | 0  | 19 | 9                       | 2                             | 0  |
|-----------|-------|--------|----|----|-------------------------|-------------------------------|----|
| add       | 80004 | 0      | 9  | 22 | 9                       | 0                             | 32 |
| lw        | 80008 | 35     | 9  | 8  |                         | 0                             |    |
| bne       | 80012 | 5      | 8  | 21 |                         | 2 (80016+2*4 [2 Instr. Away]) |    |
| addi      | 80016 | 8      | 19 | 19 |                         | 1                             |    |
| j         | 80020 |        |    |    | 20000 ? (80000/4=20000) |                               |    |
| Exit:     | 80024 | Opcode | rs | rt | rd/immediate/address    |                               |    |

Jane Liu at Lamar University

©2004 Morgan Kaufmann Publishers 43

## Big-Endian Architecture and Little-Endian Architecture

- Which bytes are most significant in *multi-byte data types*
  - Integer
  - Floating-point
  - Character
- Big-endian system
  - The **most significant value** in the sequence is stored at the **lowest storage address**
  - Mainframe computers
    - IBM mainframes
- Little-endian system
  - The **least significant value** in the sequence is stored at the **lowest storage address**
  - Modern computers including PC
- Bi-endian system
  - PowerPC

Jane Liu at Lamar University

©2004 Morgan Kaufmann Publishers 44

## Big-Endian Architecture and Little-Endian Architecture

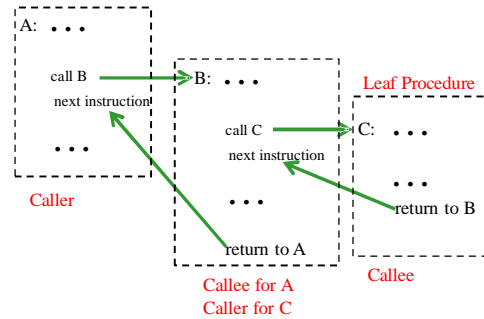
- Number 1025
    - 00000000 00000000 00000100 00000001
- | Big-Endian Addr. Representation | Little-Endian Addr. Representation |
|---------------------------------|------------------------------------|
| 00 00000000                     | 00 00000001                        |
| 01 00000000                     | 01 00000100                        |
| 10 00000100                     | 10 00000000                        |
| 11 00000001                     | 11 00000000                        |

- The terms *big-endian* and *little-endian* are derived from the *Lilliputians of Gulliver's Travels*, whose major political issue was whether soft-boiled eggs should be opened on the big side or the little side.

Jane Liu at Lamar University

©2004 Morgan Kaufmann Publishers 45

## Supporting Procedures in Computer Hardware



Jane Liu at Lamar University

©2004 Morgan Kaufmann Publishers 46

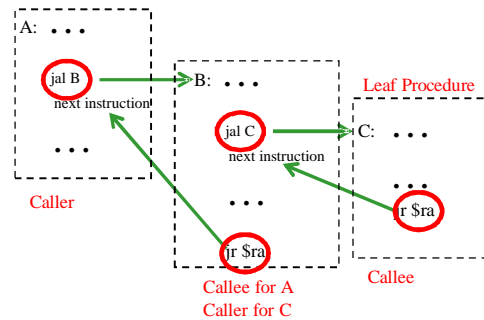
## Supporting Procedures in Computer Hardware

- Execution procedure
  - Place parameters in a place where callee can access
    - \$a0-\$a3: argument registers
  - Transfer control to procedure
    - Save address of the following instruction in \$ra
    - Jump-and-link instruction (jal)
      - jal procedure address
  - Acquire storage resources needed for callee
    - Registers or memory??
  - Perform desired task
  - Place result value in a place where caller can access
    - \$v0-\$v1: value register in which to return values
  - Return control to caller
    - Jump register instruction (jr)
      - jr \$ra

Jane Liu at Lamar University

©2004 Morgan Kaufmann Publishers 47

## Supporting Procedures in Computer Hardware



Jane Liu at Lamar University

©2004 Morgan Kaufmann Publishers 48

## Supporting Procedures in Computer Hardware

- Variables stored in registers
  - Long-live variable
    - Will be used by caller after procedure call
  - Temporary variable
    - Will not be used by caller after procedure call

```

A:  ...
    add $t2, $t1, $t0
    jal B
    sub $t4, $t2, $t3
    
```

Caller

## Supporting Procedures in Computer Hardware

- Different conventions
  - Caller-saved registers
    - Long-live variables in registers will be saved by caller in memory before procedure call
  - Callee-saved registers
    - Variables in registers will be saved by callee in memory before procedure starts
  - Caller-saved and callee-saved registers
    - Long-live variables in some registers will be saved by caller in memory before procedure call
    - Variables in some registers will be saved by callee in memory before procedure starts

## Supporting Procedures in Computer Hardware

- Different conventions
  - Caller-saved registers
    - Long-live variables in registers will be saved by caller in memory before procedure call

```

A:  ...
    add $t2, $t1, $t0
    jal B
    sub $t4, $t2, $t3
    
```

Caller

\$t0, \$t1, \$t2, \$t3, \$t4  
Which register will be saved by caller before procedure call?

## Supporting Procedures in Computer Hardware

- Different conventions
  - Callee-saved registers
    - Variables in registers will be saved by callee in memory before procedure starts

```

B:  add $t2, $t1, $t0
    jal C
    sub $t0, $t2, $t1
    jr $ra
    
```

Callee

\$t0, \$t1, \$t2, \$t3, \$t4  
Which register will be saved by callee before procedure starts?

## Supporting Procedures in Computer Hardware

- Different conventions
  - Caller-saved and callee-saved registers
    - Long-live variables in some registers will be saved by caller in memory before procedure call
    - Variables in some registers will be saved by callee in memory before procedure starts

```

A:  ...
    add $t2, $t1, $t0
    jal B
    sub $t4, $t2, $t3
    
```

Caller

```

B:  add $t2, $t1, $t0
    jal C
    sub $t0, $t2, $t1
    jr $ra
    
```

Callee

## Supporting Procedures in Computer Hardware

- MIPS convention to reduce register spilling
  - Caller-saved registers
    - \$t0-\$t9
    - \$a0-\$a3
    - \$v0-\$v1
  - Callee-saved registers
    - \$\$s0-\$\$s7
    - \$sp
    - \$ra
  - Examples (Pages 81-85)
    - Compiling a C Procedure That Doesn't call Another Procedure
    - Nested Procedures



## 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      | 26              | global pointer                               |
| \$sp      | 29              | stack pointer                                |
| \$fp      | 30              | frame pointer                                |
| \$ra      | 31              | return address                               |

Register 1 (\$at) reserved for assembler, 26-27 for operating system

### To summarize:

| MIPS operands          |                                                                              |                                                                                                                                                                                                                       |  |
|------------------------|------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--|
| Name                   | Example                                                                      | Comments                                                                                                                                                                                                              |  |
| 32 registers           | \$r0-\$r7, \$t0-\$t9, \$s0-\$s7, \$a0-\$a3, \$v0-\$v1, \$w0-\$w1, \$f0-\$f31 | Final locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register labels always equate to Register Set A                                                                              |  |
| 2 <sup>32</sup> Memory | Memory[0], Memory[4], ..., Memory[20487292]                                  | Accessed only by data transfer instructions. MIPS uses byte addresses, so hexadecimal words (like 0x4) Memory holds data structures, such as arrays. Not subject to access, such as a write ahead on sequential cache |  |

| MIPS assembly language |                         |                      |                                         |                                   |
|------------------------|-------------------------|----------------------|-----------------------------------------|-----------------------------------|
| Category               | Instruction             | Example              | Meaning                                 | Comments                          |
| Arithmetic             | add                     | add \$s1, \$s2, \$s3 | $Ss1 = Ss2 + Ss3$                       | Three operands; data in registers |
|                        | subtract                | sub \$s1, \$s2, \$s3 | $Ss1 = Ss2 - Ss3$                       | Three operands; data in registers |
|                        | add immediate           | addi \$s1, \$s0, 100 | $Ss1 = Ss0 + 100$                       | loads in add immediates           |
| Data transfer          | load word               | lw \$s1, 10(\$s2)    | $Ss1 = Memory[Ss2 + 100]$               | Word from memory to register      |
|                        | store word              | sw \$s1, 10(\$s2)    | $Memory[Ss2 + 100] = Ss1$               | Word from register to memory      |
|                        | load byte               | lb \$s1, 10(\$s2)    | $Ss1 = Memory[Ss2 + 100]$               | Byte from memory to register      |
| Conditional branch     | branch on equal         | beq \$s1, \$s2, 25   | $\{Ss1 == Ss2\} ? pc to PC + 4 + 100$   | Equal test; PC-relative branch    |
|                        | branch on not equal     | bne \$s1, \$s2, 25   | $\{Ss1 != Ss2\} ? pc to PC + 4 + 100$   | Not equal test; PC-relative       |
|                        | set on less than        | slt \$s1, \$s2, \$s3 | $\{Ss2 < Ss3\} ? Ss1 = 1, else Ss1 = 0$ | Compare less than; for loop, true |
| Control flow           | set less than immediate | slti \$s1, \$s2, 100 | $\{Ss2 < 100\} ? Ss1 = 1, else Ss1 = 0$ | Compare less than constant        |
|                        | jump                    | j 25(0)              | pc to 25(0)                             | unconditional address             |
|                        | jump and link           | jal 25(0)            | pc to 25(0), \$ra to 25(0)              | For return, store link return     |

## Chapter 2 - 4

### Instruction Set Architecture (ISA)

- Interface between software and hardware
  - Abstraction layer

### ISA

- Complex instruction set computer (CISC)
- Reduced instruction set computer (RISC)
- Explicitly parallel instruction computing design (EPIC)

## ISA

- **Complex instruction set computer (CISC)**
  - **Compiler and assembler technology is immature**
  - **Hundreds of instructions to make programming easier**
    - Different width to save memory space
      - 8-bit – 120-bit

## ISA

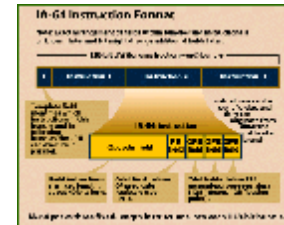
- **Reduced instruction set computer (RISC)**
  - **RAM**
    - Larger and larger
    - Memory conservation less concern
    - CPU design shifted to raw performance
  - **Small number of relatively simple, fixed-length instructions, always 32 bits long**
    - Easier and faster to execute
    - Require less transistors
    - Better performance
    - Make it easier to design Superscalar processors
      - Instruction Level Parallelism (ILP)
      - Superscalar
        - Execute more than one instruction at a time
        - Increase design complexity
  - **Processors**
    - MIPS Rxxx
    - Sun SPARC
    - IBM and Motorola PowerPC
    - ARM processors
    - Alpha

## ISA

- **Explicitly parallel instruction computing design (EPIC)**
  - **To simplify process design**
    - Do not need most of the complex control circuitry that superscalar chips must use to coordinate parallel execution at runtime
  - **Combine two or more instructions into a single bundle or packet**
    - Quickly execute instructions in parallel
    - Code expansion
      - programs grow larger
  - **Move burden to compiler**
    - Smart compilation
  - **Widely considered as a successful replacement for RISC**
  - **Intel IA-64 based on EPIC**

## IA-64 Instruction Bundle Formats

- 128-bit wide
- VLIW
- 3 instruction slots
  - Each 41-bit wide
- A 5-bit template field to allocate instructions into the slot thus a total of 32 possible combinations



## IA-64 Instruction Bundle Formats

- **I - ALU and non-ALU Integer**
  - Include the usual integer arithmetic/logical operations, together with various moves, tests and shifts.
- **M - ALU and Memory**
  - Include integer arithmetic/logical operations, together with memory access
- **F - Floating point**
  - For floating point operations
- **B - Branches**
  - Branches and procedure calls
  - Contain static prediction information which is used until the branch prediction algorithm builds up enough history information.
- **L+X - Extended**
  - Various other things

## IA-64 Bundle Template

