## Example of multiple operands

- Instructions may have 3, 2, 1, or 0 operands
- Number of operands may affect instruction length
- Operand order is fixed (destination first, but need not that way)

```
add $s0, $s1, $s2    ; Add $s2 and $s1 and store result in $s0

add $s0, $s1         ; Add $s1 and $s0 and store result in $s0

add $s0              ; Add contents of a fixed location to $s0

add                  ; Add two fixed locations and store result
```



## Where operands are stored

- Memory locations
  - Instruction includes address of location
- Registers
  - Instruction includes register number
- Stack location
  - Instruction opcode implies that the operand is in stack
- Fixed register
  - Like accumulator, or depends on inst
  - Hi and Lo register in MIPS
- Fixed location
  - Default operands like interrupt vectors



## 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;
```

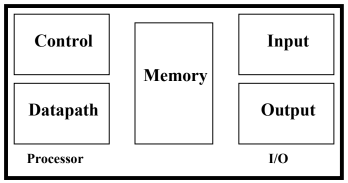
MIPS code:

```
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?
  - More register will slow register file down.

## Registers vs. Memory

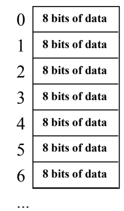
- Arithmetic instructions operands must be registers,
  — only 32 registers provided
- Compiler associates variables with registers
- What about programs with lots of variables






## Memory Organization

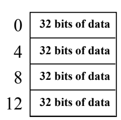
- Viewed as a large, single-dimension array, with an address.
- A memory address is an index into the array
- "Byte addressing" means that the index points to a byte of memory.






## Memory Organization

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



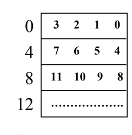

Registers hold 32 bits of data

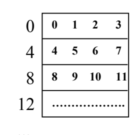
- $2^{32}$ bytes with byte addresses from 0 to $2^{32}$-1
- $2^{30}$ words with byte addresses 0, 4, 8, ... $2^{32}$-4
- Words are aligned
  i.e., what are the least 2 significant bits of a word address?



## Addressing within a word

- Each word has four bytes
- Which byte is first and which is last
- Two Choices
  - Least significant byte is byte “0” -> Little Endian
  - Most significant byte is byte “0” -> Big Endian



## 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!



## Addressing

- Memory address for load and store has two parts
  - A register whose content is known
  - An offset stored in 16 bits
- The offset can be positive or negative
  - It is written in terms of number of bytes
  - It is but in instruction in terms of number of words
  - 32 byte offset is written as 32 but stored as 8
- Address is content of register + offset
- All addresses have both these components
- If no register needs to be used then use register 0
  - Register 0 always stores value 0
- If no offset, then offset is 0



## 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

- 

| <u>Instruction</u> | <u>Meaning</u> |
|---|---|
| `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 $t0, $s1, $s2`
  - registers have numbers, `$t0=8, $s1=17, $s2=18`
- Instruction Format:

| | | | | | |
|---|---|---|---|---|---|
| 000000 | 10001 | 10010 | 01000 | 00000 | 100000 |

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

- *Can you guess what the field names stand for?*



## 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)`

| | | | |
|---|---|---|---|
| 35 | 18 | 8 | 32 |

| op | rs | rt | 16 bit number |
|---|---|---|---|

- Where's the compromise?



## 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
beq $t0, $t1, Label
```

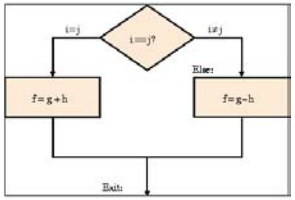
- Example: if (i==j) h = i + j;

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



## Conditional Execution

- A simple conditional execution
- Depending on i==j or i!=j, result is different



## Instruction Sequencing

- **MIPS unconditional branch instructions:**

```
j  label
```

- **Example:**
  **f, g, and h are in registers $s3, $s4, and $s5**

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

- *Can you build a simple for loop?*



## 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:**

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



## Control Flow

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

```
                         if  $s1 < $s2 then
                            $t0 = 1
slt $t0, $s1, $s2        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**



## 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?**

## Supporting Procedures

- How to make procedures work?
- How to make their implementation efficient?

- Transfer controls to callee and back to caller
  `jal procedure_address`
  `jr  $ra`
- Pass parameters and results
  `$a0-$a3, $v0-$v1, $ra, stack`
- Acquire local storage
  Use stack: `$sp, $fp`
- Preserve and restore caller's context



## 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 |



## Other Issues

- Things we are not going to cover
  support for procedures
  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.



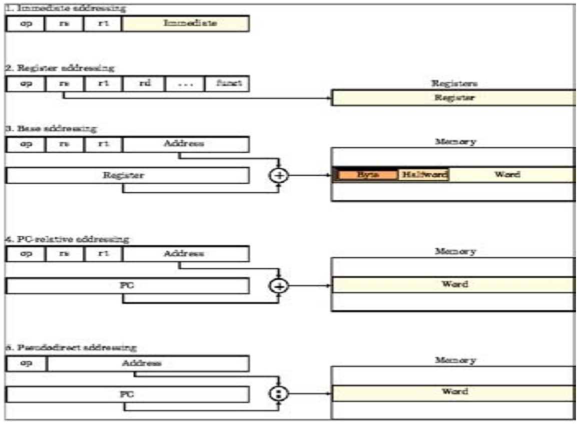
## Overview of MIPS

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

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

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

## Various Addressing Modes






## 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:

| | | | | |
|---|---|---|---|---|
| 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?



## 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:

| | | | | |
|---|---|---|---|---|
| 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



## 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 |
| Unconditional 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 |