# Chapter 2

## Instructions: Language of the Computer

# Instructions:

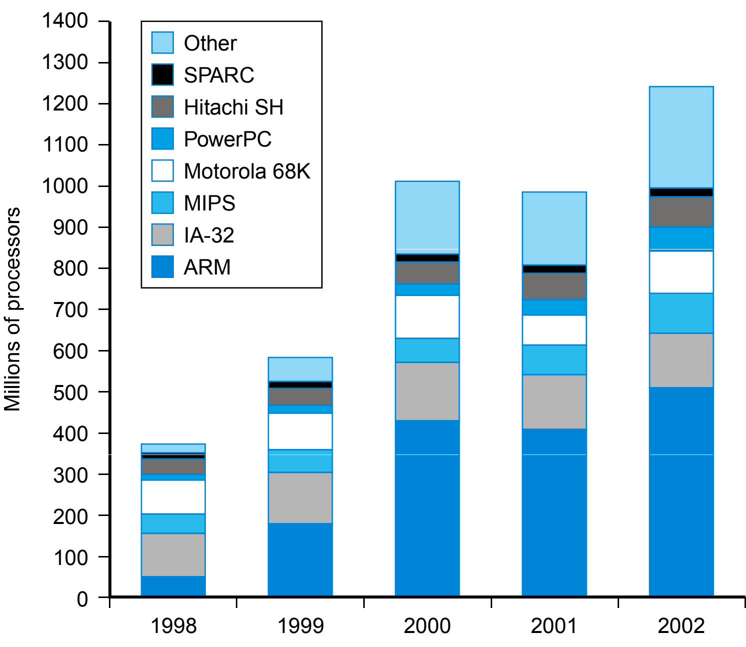
- Language of the Machine
- More primitive than higher level languages
  - e.g., no sophisticated control flow
- Very restrictive
  - e.g., MIPS Arithmetic Instructions

# 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*

# Instructions:



# Instruction Set

- The repertoire of instructions of a computer
- Different computers have different instruction sets
  - But with many aspects in common
- Early computers had very simple instruction sets
  - Simplified implementation
- Many modern computers also have simple instruction sets

# The MIPS Instruction Set

- Used as the example throughout the book
- Stanford MIPS commercialized by MIPS Technologies (www.mips.com)
- Large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …
- Typical of many modern ISAs
  - See MIPS Reference Data tear-out card, and Appendixes B and E

# 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)

*"The natural number of operands for an operation like addition is three…requiring every instruction to have exactly three operands, no more and no less, conforms to the philosophy of keeping the hardware simple"*

# 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
```

# MIPS arithmetic

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

# Arithmetic Operations

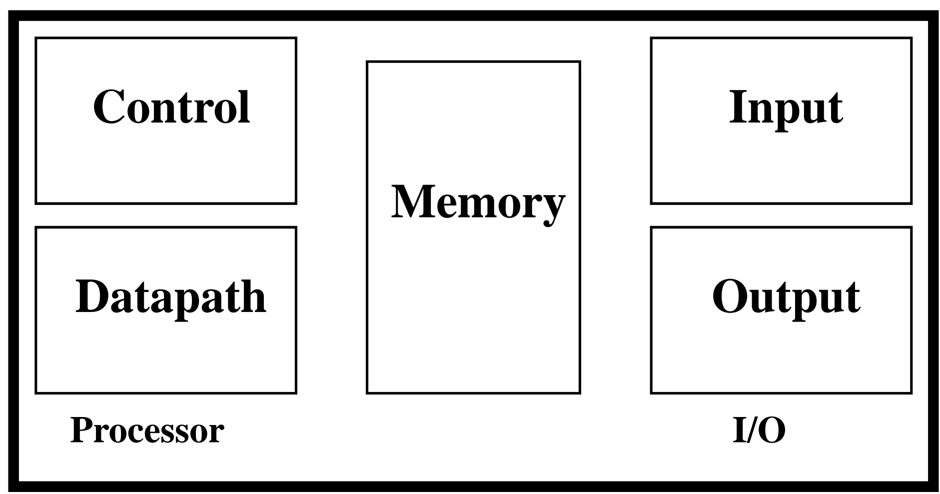
- Add and subtract, three operands
  - Two sources and one destination

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

- All arithmetic operations have this form
- *Design Principle 1:* Simplicity favors regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

# 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



# Register Operands

- Arithmetic instructions use register operands
- MIPS has a 32 × 32-bit register file
  - Use for frequently accessed data
  - Numbered 0 to 31
  - 32-bit data called a "word"
- Assembler names
  - $t0, $t1, …, $t9 for temporary values
  - $s0, $s1, …, $s7 for saved variables
- *Design Principle 2:* Smaller is faster
  - main memory: millions of locations

# Register Operand Example

- C code:

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

  - f, …, j in $s0, …, $s4

- Compiled MIPS code:

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

# Registers vs. Memory

- 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!

# 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.

| Address | Data |
|---|---|
| 0 | 8 bits of data |
| 1 | 8 bits of data |
| 2 | 8 bits of data |
| 3 | 8 bits of data |
| 4 | 8 bits of data |
| 5 | 8 bits of data |
| 6 | 8 bits of data |
| ... | |

# Memory Organization

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

| Address | Data |
| --- | --- |
| 0 | 32 bits of data |
| 4 | 32 bits of data |
| 8 | 32 bits of data |
| 12 | 32 bits of data |
| ... | |

**Registers hold 32 bits of data**

# 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?

# Memory Operands

- Main memory used for composite data
  - Arrays, structures, dynamic data
- To apply arithmetic operations
  - 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 at least address of a word
  - Little Endian: least-significant byte at least address

# Memory Operand Example 1

- C code:

  `g = h + A[8];`

  - g in $s1, h in $s2, base address of A in $s3
- Compiled MIPS code:
  - Index 8 requires offset of 32
    - 4 bytes per word

```
lw  $t0, 32($s3)    # load word
add $s1, $s2, $t0
```

offset

base register

# Memory Operand Example 2

- C code:

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

  - h in $s2, base address of A in $s3
- Compiled MIPS code:
  - Index 8 requires offset of 32

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

# Immediate Operands

- Constant data specified in an instruction

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

- No subtract immediate instruction
  - Just use a negative constant

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

- *Design Principle 3:* Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction

# The Constant Zero

- MIPS register 0 ($zero) is the constant 0
  - Cannot be overwritten
- Useful for common operations
  - E.g., move between registers

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

# Unsigned Binary Integers

- $$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$
- Range: 0 to $+2^n - 1$
- Example
  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
    $= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$
- Using 32 bits
  - 0 to +4,294,967,295

# 2s-Complement Signed Integers

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

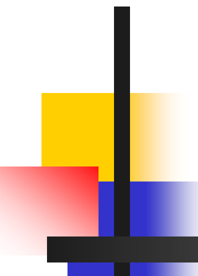
- Range: $-2^{n-1}$ to $+2^{n-1} - 1$
- Example
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^2 + 0\times2^1 + 0\times2^0$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$
- Using 32 bits
  - –2,147,483,648 to +2,147,483,647

# 2s-Complement Signed Integers

- Bit 31 is sign bit
  - 1 for negative numbers
  - 0 for non-negative numbers
- $-(-2^{n-1})$ can't be represented
- Non-negative numbers have the same unsigned and 2s-complement representation
- Some specific numbers
  - 0: 0000 0000 … 0000
  - –1: 1111 1111 … 1111
  - Most-negative: 1000 0000 … 0000
  - Most-positive: 0111 1111 … 1111
- Negation: Complement and add 1
  - Complement means $1 \rightarrow 0$, $0 \rightarrow 1$

# Sign Extension

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



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

# 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

| 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` |

The College of New Jersey

# Machine Language

- 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 | 01001 | 00000 | 100000 |
|---|---|---|---|---|---|
| **op** | **rs** | **rt** | **rd** | **shamt** | **funct** |

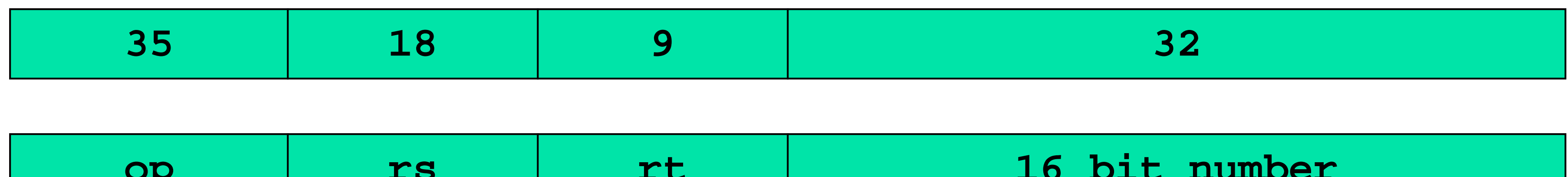
- *Can you guess what the field names stand for?*

# 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

# Machine Language

- 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?

# Representing Instructions

- Instructions are encoded in binary
  - Called machine code
- MIPS instructions
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, …
  - Regularity!
- Register numbers
  - $t0 – $t7 are reg's 8 – 15
  - $t8 – $t9 are reg's 24 – 25
  - $s0 – $s7 are reg's 16 – 23

# MIPS R-format Instructions

| op | rs | rt | rd | shamt | funct |
|---|---|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |

- 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 Example

| op | rs | rt | rd | shamt | funct |
|---|---|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |

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

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

$00000010001100100100000000100000_2 = 02324020_{16}$

# MIPS I-format Instructions

| op | rs | rt | constant or address |
|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 16 bits |

- 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
- *Design Principle 4:* Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

# Logical Operations

- 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

# Shift Operations

| op | rs | rt | rd | shamt | funct |
|---|---|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |

- shamt: 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)

# AND Operations

- 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 Operations

- 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 1101 1100 0000 |

# NOT Operations

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

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

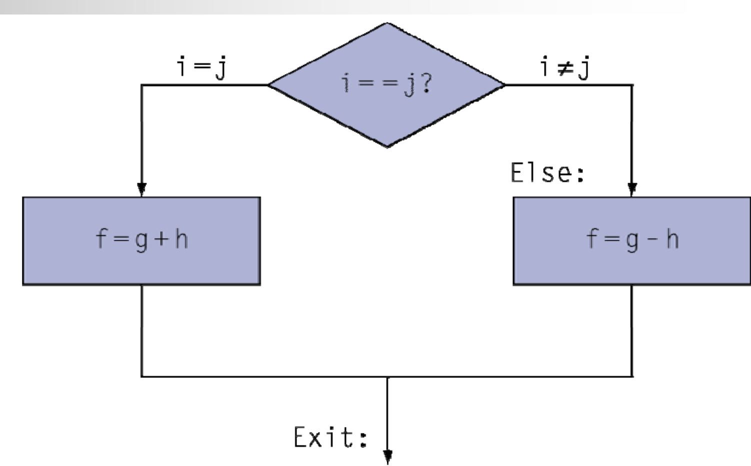
Register 0: always read as zero

| | |
|---|---|
| $t1 | 0000 0000 0000 0000 0011 1100 0000 0000 |
| $t0 | 1111 1111 1111 1111 1100 0011 1111 1111 |

# Conditional Operations

- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- `beq rs, rt, L1`
  - if (rs == rt) branch to instruction labeled L1;
- `bne rs, rt, L1`
  - if (rs != rt) branch to instruction labeled L1;
- `j L1`
  - unconditional jump to instruction labeled L1

# Compiling If Statements




- C code:

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

  - f, g, … in $s0, $s1, …

- Compiled MIPS code:

Assembler calculates addresses

```
      bne $s3, $s4, Else
      add $s0, $s1, $s2
      j   Exit
Else: sub $s0, $s1, $s2
Exit: …
```

# Compiling Loop Statements

- C code:

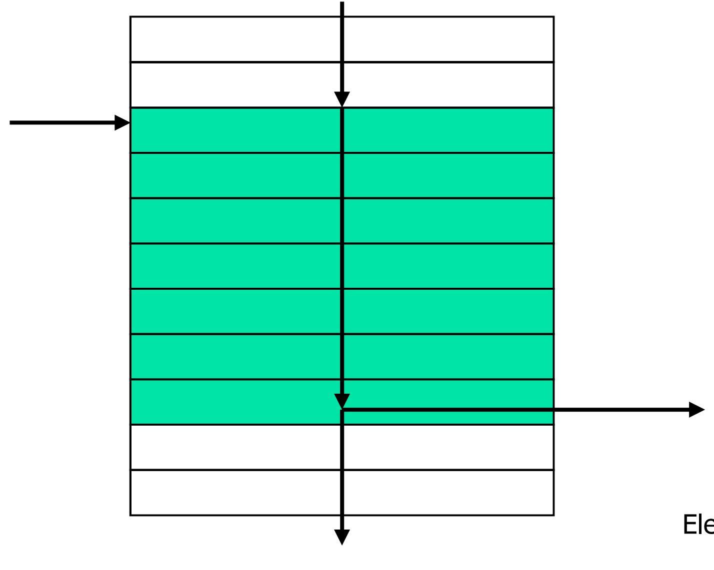
```
while (save[i] == k) i += 1;
```

  - i in $s3, k in $s5, address of save in $s6
- 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 identifies basic blocks for optimization
- An advanced processor can accelerate execution of basic blocks

# More Conditional Operations

- Set result to 1 if a condition is true
  - Otherwise, set to 0
- `slt rd, rs, rt`
  - if (rs < rt) rd = 1; else rd = 0;
- `slti rt, rs, constant`
  - if (rs < constant) rt = 1; else rt = 0;
- Use in combination with beq, bne

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

# Branch Instruction Design

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

# Signed vs. Unsigned

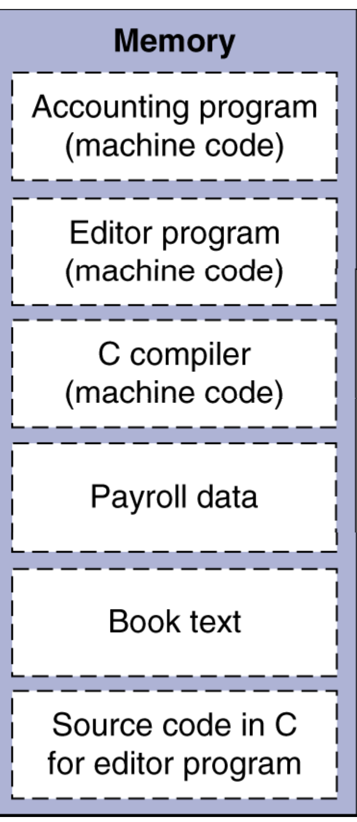
- Signed comparison: `slt`, `slti`
- Unsigned comparison: `sltu`, `sltui`
- Example
  - \$s0 = 1111 1111 1111 1111 1111 1111 1111 1111
  - \$s1 = 0000 0000 0000 0000 0000 0000 0000 0001
  - `slt  $t0, $s0, $s1  # signed`
    - $-1 < +1 \Rightarrow \$t0 = 1$
  - `sltu $t0, $s0, $s1  # unsigned`
    - $+4{,}294{,}967{,}295 > +1 \Rightarrow \$t0 = 0$

# Procedure Calling

- Steps required
  1. Place parameters in registers
  2. Transfer control to procedure
  3. Acquire storage for procedure
  4. Perform procedure's operations
  5. Place result in register for caller
  6. Return to place of call

# Stored Program Computers

## The BIG Picture




- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
  - e.g., compilers, linkers, …
- Binary compatibility allows compiled programs to work on different computers
  - Standardized ISAs

# Stored Program Concept

- 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

# Control

- 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: ....
```

# 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?*

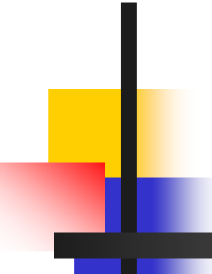
# So far:

- Instruction / Meaning

| 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 number | | |
| **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

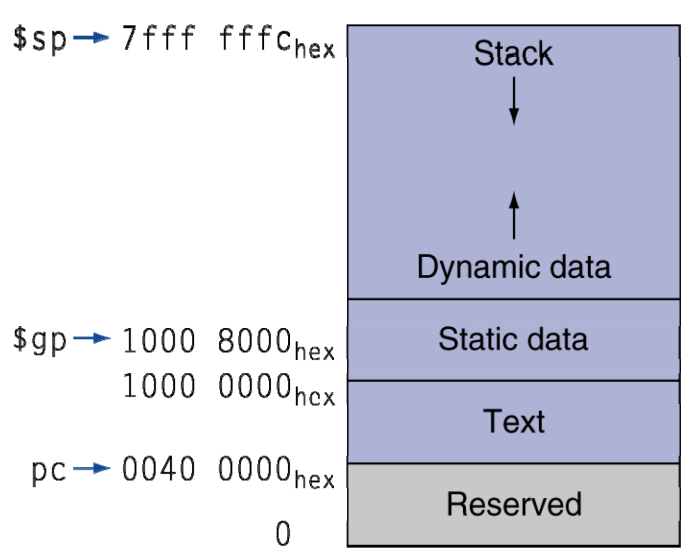
# Policy of Use Conventions

| Name | Register number | Usage | Preserved on call? |
|---|---|---|---|
| `$zero` | 0 | the constant value 0 | n.a. |
| `$v0-$v1` | 2-3 | values for results and expression evaluation | no |
| `$a0-$a3` | 4-7 | arguments | yes |
| `$t0-$t7` | 8-15 | temporaries | no |
| `$s0-$s7` | 16-23 | saved | yes |
| `$t8-$t9` | 24-25 | more temporaries | no |
| `$gp` | 28 | global pointer | yes |
| `$sp` | 29 | stack pointer | yes |
| `$fp` | 30 | frame pointer | yes |
| `$ra` | 31 | return address | yes |

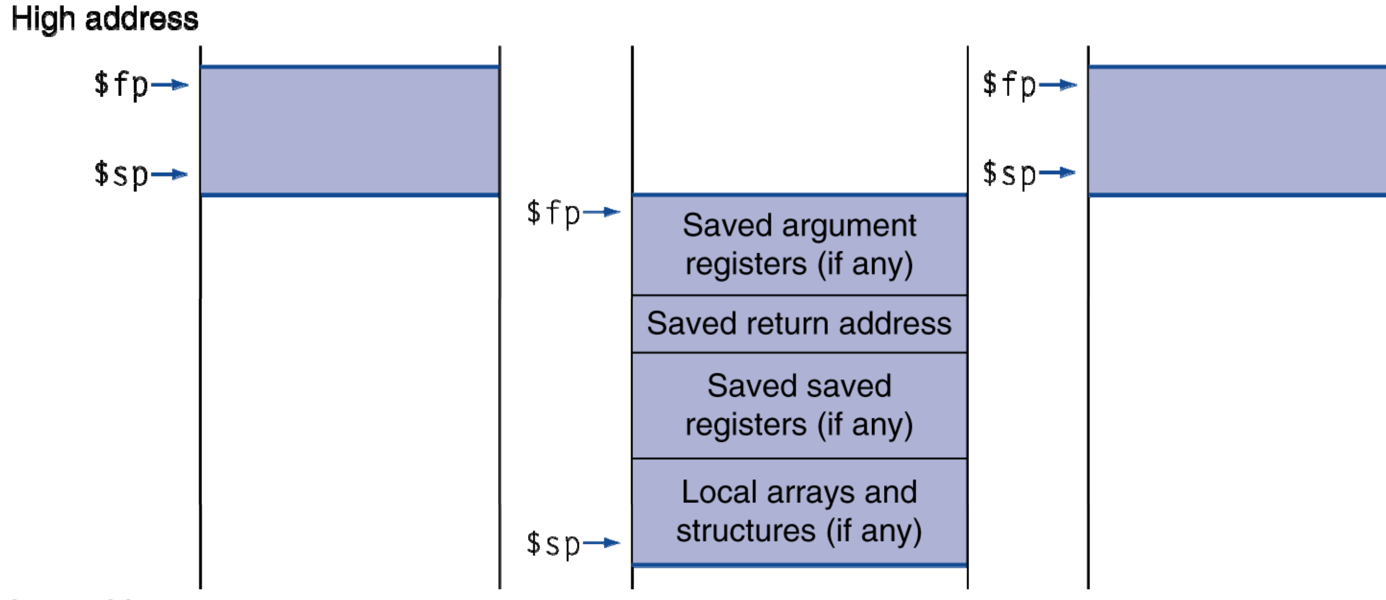
Register 1 ($at) reserved for assembler, 26-27 for operating system

# Memory Layout

- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
  - $gp initialized to address allowing ±offsets into this segment
- Dynamic data: heap
  - E.g., malloc in C, new in Java
- Stack: automatic storage



# Local Data on the Stack




- Local data allocated by callee
  - e.g., C automatic variables
- Procedure frame (activation record)
  - Used by some compilers to manage stack storage

# Procedure Call Instructions

- Procedure call: jump and link

  `jal ProcedureLabel`

  - Address of following instruction put in $ra
  - Jumps to target address
- Procedure return: jump register

  `jr $ra`

  - Copies $ra to program counter
  - Can also be used for computed jumps
    - e.g., for 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;
}
```

  - 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
```

Save $s0 on stack

Procedure body

Result

Restore $s0

Return

# 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 f;
  else return n * fact(n - 1);
}
```

  - Argument n in $a0
  - Result in $v0

# Non-Leaf Procedure Example

The College of New Jersey

- 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
    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
```

# Character Data

- Byte-encoded character sets
  - ASCII: 128 characters
    - 95 graphic, 33 control
  - Latin-1: 256 characters
    - ASCII, +96 more graphic characters
- Unicode: 32-bit character set
  - Used in Java, C++ wide characters, …
  - Most of the world's alphabets, plus symbols
  - UTF-8, UTF-16: variable-length encodings

# Byte/Halfword Operations

- Could use bitwise operations
- MIPS byte/halfword load/store
  - String processing is a common case

```
lb rt, offset(rs)     lh rt, offset(rs)
```

  - Sign extend to 32 bits in rt

```
lbu rt, offset(rs)    lhu rt, offset(rs)
```

  - Zero extend to 32 bits in rt

```
sb rt, offset(rs)     sh rt, offset(rs)
```

  - Store just rightmost byte/halfword

# String Copy Example

- C code:
  - Null-terminated string

```
void strcpy (char x[], char y[])
{ int i;
  i = 0;
  while ((x[i]=y[i])!='\0')
    i += 1;
}
```

  - Addresses of x, y in $a0, $a1
  - i in $s0

# String Copy Example

- MIPS code:

```
strcpy:
     addi $sp, $sp, -4        # adjust stack for 1 item
     sw   $s0, 0($sp)         # save $s0
     add  $s0, $zero, $zero   # i = 0
L1:  add  $t1, $s0, $a1       # addr of y[i] in $t1
     lbu  $t2, 0($t1)         # $t2 = y[i]
     add  $t3, $s0, $a0       # addr of x[i] in $t3
     sb   $t2, 0($t3)         # x[i] = y[i]
     beq  $t2, $zero, L2      # exit loop if y[i] == 0
     addi $s0, $s0, 1         # i = i + 1
     j    L1                  # next iteration of loop
L2:  lw   $s0, 0($sp)         # restore saved $s0
     addi $sp, $sp, 4         # pop 1 item from stack
     jr   $ra                 # and return
```

# 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.

# Constants

- MIPS Instructions:

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

- How do we make this work?

- 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
```

**filled with zeros**

| 1010101010101010 | 0000000000000000 |
|---|---|

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

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

| | | |
|---|---|---|
| | 1010101010101010 | 0000000000000000 |
| ori | 0000000000000000 | 1010101010101010 |
| | 1010101010101010 | 1010101010101010 |

# Branch Addressing

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

| op | rs | rt | constant or address |
|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 16 bits |

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

# Jump Addressing

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

| op | address |
|---|---|
| 6 bits | 26 bits |

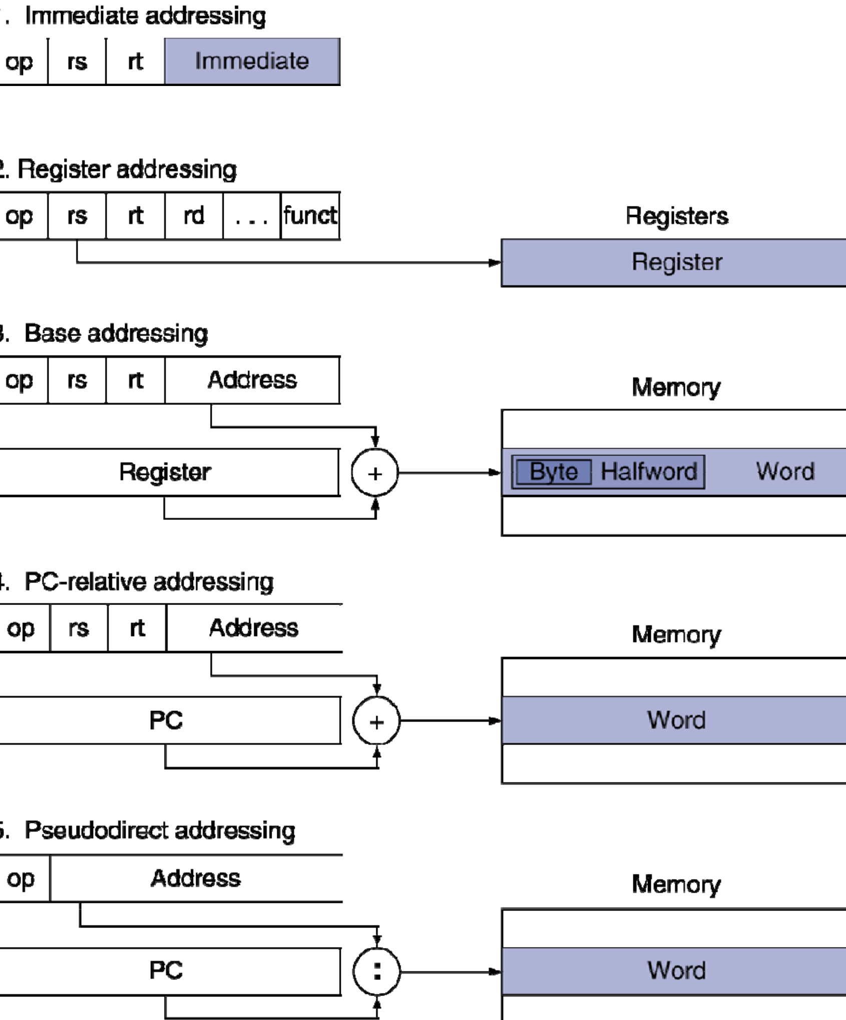
- (Pseudo)Direct jump addressing
  - Target address = $PC_{31\ldots28}$ : (address × 4)

# 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



# Synchronization

- Two processors sharing an area of memory
  - P1 writes, then P2 reads
  - Data race if P1 and P2 don't synchronize
    - Result depends of order of accesses
- Hardware support required
  - Atomic read/write memory operation
  - No other access to the location allowed between the read and write
- Could be a single instruction
  - E.g., atomic swap of register ↔ memory
  - Or an atomic pair of instructions

# Synchronization in MIPS

- Load linked: `ll rt, offset(rs)`
- Store conditional: `sc rt, offset(rs)`
  - Succeeds if location not changed since the `ll`
    - Returns 1 in rt
  - Fails if location is changed
    - Returns 0 in rt
- Example: atomic swap (to test/set lock variable)

```
try: add $t0,$zero,$s4 ;copy exchange value
     ll  $t1,0($s1)    ;load linked
     sc  $t0,0($s1)    ;store conditional
     beq $t0,$zero,try ;branch store fails
     add $s4,$zero,$t1 ;put load value in $s4
```

# Translation and Startup

C program → Compiler → Assembly language program → Assembler → Object: Machine language module + Object: Library routine (machine language) → Linker → Executable: Machine language program → Loader → Memory

Many compilers produce object modules directly

Static linking

# Assembler Pseudoinstructions

- Most assembler instructions represent machine instructions one-to-one
- Pseudoinstructions: figments of the assembler's imagination

```
move $t0, $t1      →  add $t0, $zero, $t1
blt $t0, $t1, L    →  slt $at, $t0, $t1
                      bne $at, $zero, L
```

  - $at (register 1): assembler temporary

# Producing an Object Module

- Assembler (or compiler) translates program into machine instructions
- Provides information for building a complete program from the pieces
  - Header: described contents of object module
  - Text segment: translated instructions
  - Static data segment: data allocated for the life of the program
  - Relocation info: for contents that depend on absolute location of loaded program
  - Symbol table: global definitions and external refs
  - Debug info: for associating with source code

# Linking Object Modules

- Produces an executable image
  1. Merges segments
  2. Resolve labels (determine their addresses)
  3. Patch location-dependent and external refs
- Could leave location dependencies for fixing by a relocating loader
  - But with virtual memory, no need to do this
  - Program can be loaded into absolute location in virtual memory space

# Loading a Program

- Load from image file on disk into memory
  1. Read header to determine segment sizes
  2. Create virtual address space
  3. Copy text and initialized data into memory
     - Or set page table entries so they can be faulted in
  4. Set up arguments on stack
  5. Initialize registers (including $sp, $fp, $gp)
  6. Jump to startup routine
     - Copies arguments to $a0, … and calls main
     - When main returns, do exit syscall

# Dynamic Linking

- Only link/load library procedure when it is called
  - Requires procedure code to be relocatable
  - Avoids image bloat caused by static linking of all (transitively) referenced libraries
  - Automatically picks up new library versions

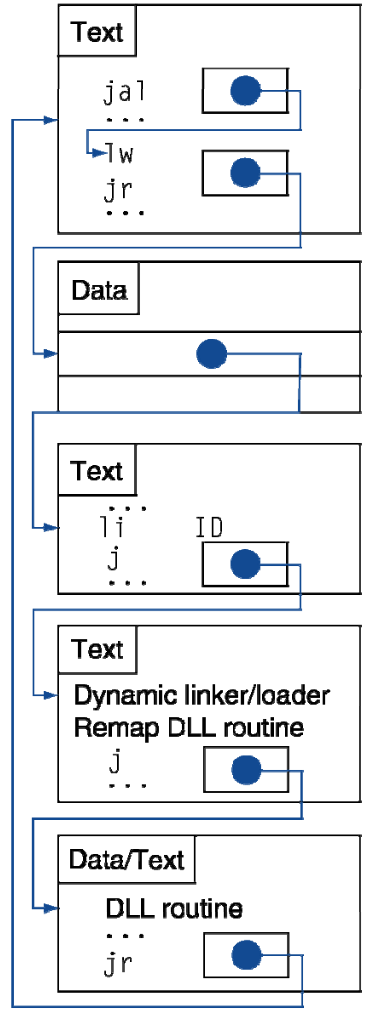
# Lazy Linkage

Indirection table

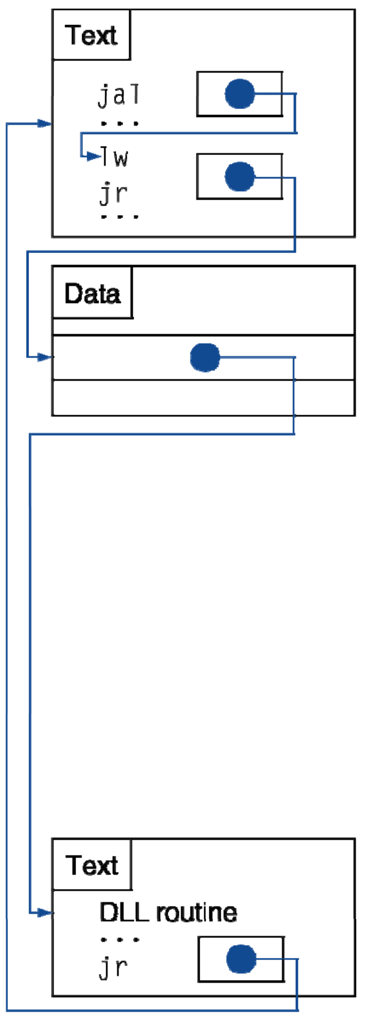
Stub: Loads routine ID,
Jump to linker/loader

Linker/loader code

Dynamically
mapped code




a. First call to DLL routine




b. Subsequent calls to DLL routine

# Starting Java Applications

Java program

Compiler

Class files (Java bytecodes)

Java library routines (machine language)

Just In Time compiler

Java Virtual Machine

Compiled Java methods (machine language)

Simple portable instruction set for the JVM

Compiles bytecodes of “hot” methods into native code for host machine

Interprets bytecodes

# C Sort Example

- Illustrates use of assembly instructions for a C bubble sort function
- Swap procedure (leaf)

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

  - v in $a0, k in $a1, temp in $t0

# The Procedure Swap

```
swap: sll $t1, $a1, 2   # $t1 = k * 4
      add $t1, $a0, $t1 # $t1 = v+(k*4)
                        #   (address of v[k])
      lw $t0, 0($t1)    # $t0 (temp) = v[k]
      lw $t2, 4($t1)    # $t2 = v[k+1]
      sw $t2, 0($t1)    # v[k] = $t2 (v[k+1])
      sw $t0, 4($t1)    # v[k+1] = $t0 (temp)
      jr $ra            # return to calling
  routine
```

# The Sort Procedure in C

- Non-leaf (calls swap)

```
void sort (int v[], int n)
{
  int i, j;
  for (i = 0; i < n; i += 1) {
    for (j = i – 1;
         j >= 0 && v[j] > v[j + 1];
         j -= 1) {
      swap(v,j);
    }
  }
}
```

  - v in $a0, k in $a1, i in $s0, j in $s1

# The Procedure Body

```
         move $s2, $a0          # save $a0 into $s2
         move $s3, $a1          # save $a1 into $s3
         move $s0, $zero        # i = 0
for1tst: slt  $t0, $s0, $s3     # $t0 = 0 if $s0 ≥ $s3 (i ≥ n)
         beq  $t0, $zero, exit1 # go to exit1 if $s0 ≥ $s3 (i ≥ n)
         addi $s1, $s0, –1      # j = i – 1
for2tst: slti $t0, $s1, 0       # $t0 = 1 if $s1 < 0 (j < 0)
         bne  $t0, $zero, exit2 # go to exit2 if $s1 < 0 (j < 0)
         sll  $t1, $s1, 2       # $t1 = j * 4
         add  $t2, $s2, $t1     # $t2 = v + (j * 4)
         lw   $t3, 0($t2)       # $t3 = v[j]
         lw   $t4, 4($t2)       # $t4 = v[j + 1]
         slt  $t0, $t4, $t3     # $t0 = 0 if $t4 ≥ $t3
         beq  $t0, $zero, exit2 # go to exit2 if $t4 ≥ $t3
         move $a0, $s2          # 1st param of swap is v (old $a0)
         move $a1, $s1          # 2nd param of swap is j
         jal  swap              # call swap procedure
         addi $s1, $s1, –1      # j –= 1
         j    for2tst           # jump to test of inner loop
exit2:   addi $s0, $s0, 1       # i += 1
         j    for1tst           # jump to test of outer loop
```

Move params

Outer loop

Inner loop

Pass params & call

Inner loop

Outer loop

# The Full Procedure

```
sort:    addi $sp,$sp, -20       # make room on stack for 5 registers
         sw $ra, 16($sp)         # save $ra on stack
         sw $s3,12($sp)          # save $s3 on stack
         sw $s2, 8($sp)          # save $s2 on stack
         sw $s1, 4($sp)          # save $s1 on stack
         sw $s0, 0($sp)          # save $s0 on stack
         …                       # procedure body
         …
         exit1: lw $s0, 0($sp)   # restore $s0 from stack
         lw $s1, 4($sp)          # restore $s1 from stack
         lw $s2, 8($sp)          # restore $s2 from stack
         lw $s3,12($sp)          # restore $s3 from stack
         lw $ra,16($sp)          # restore $ra from stack
         addi $sp,$sp, 20        # restore stack pointer
         jr $ra                  # return to calling routine
```

# Effect of Compiler Optimization

Compiled with gcc for Pentium 4 under Linux

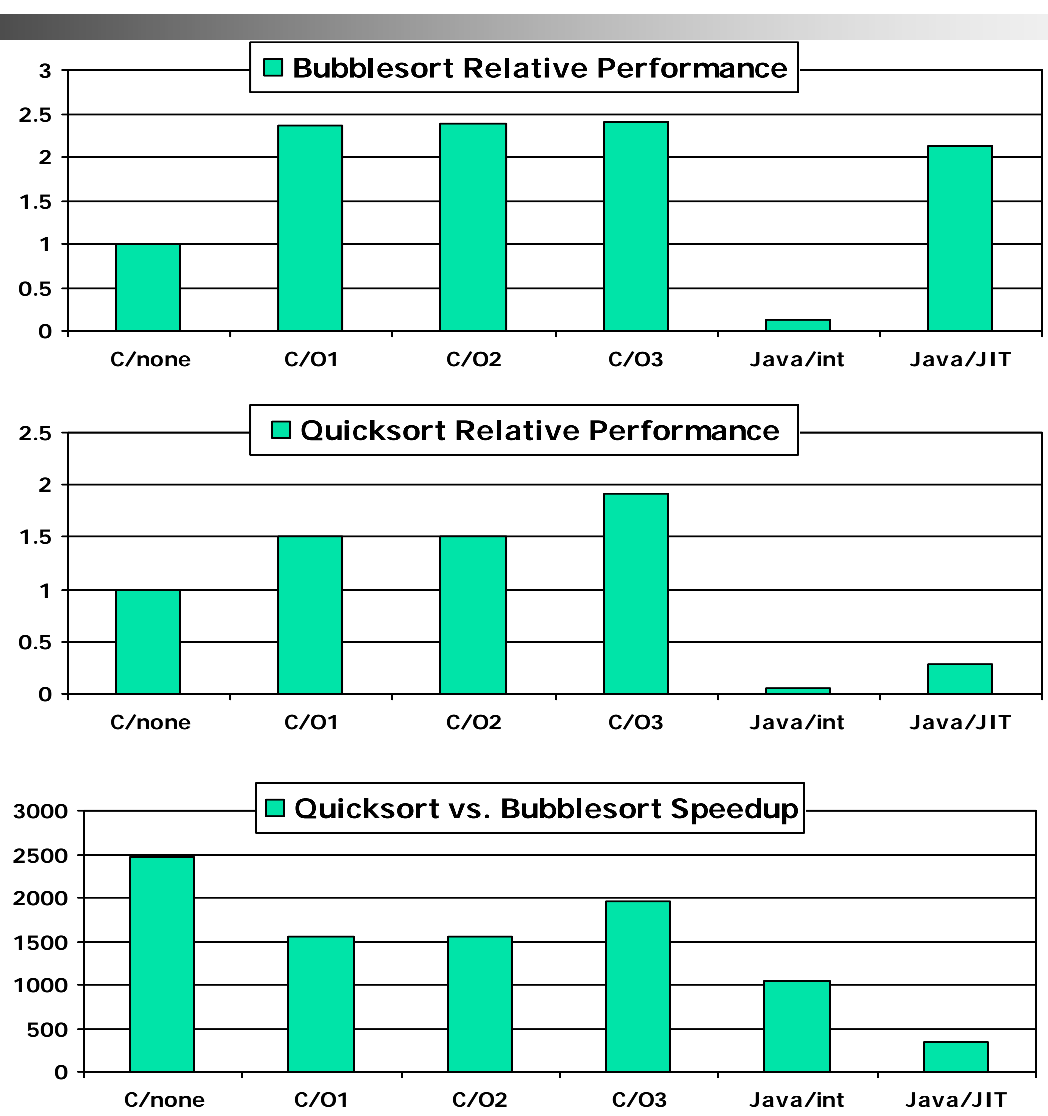
Relative Performance
3
2.5
2
1.5
1
0.5
0
none
O1
O2
O3

Instruction count
140000
120000
100000
80000
60000
40000
20000
0
none
O1
O2
O3

Clock Cycles
180000
160000
140000
120000
100000
80000
60000
40000
20000
0
none
O1
O2
O3

CPI
2
1.5
1
0.5
0
none
O1
O2
O3

# Effect of Language and Algorithm



# Lessons Learnt

- Instruction count and CPI are not good performance indicators in isolation
- Compiler optimizations are sensitive to the algorithm
- Java/JIT compiled code is significantly faster than JVM interpreted
  - Comparable to optimized C in some cases
- Nothing can fix a dumb algorithm!

# Arrays vs. Pointers

- Array indexing involves
  - Multiplying index by element size
  - Adding to array base address
- Pointers correspond directly to memory addresses
  - Can avoid indexing complexity

# Comparison of Array vs. Ptr

- Multiply “strength reduced” to shift
- Array version requires shift to be inside loop
  - Part of index calculation for incremented i
  - c.f. incrementing pointer
- Compiler can achieve same effect as manual use of pointers
  - Induction variable elimination
  - Better to make program clearer and safer

# ARM & MIPS Similarities

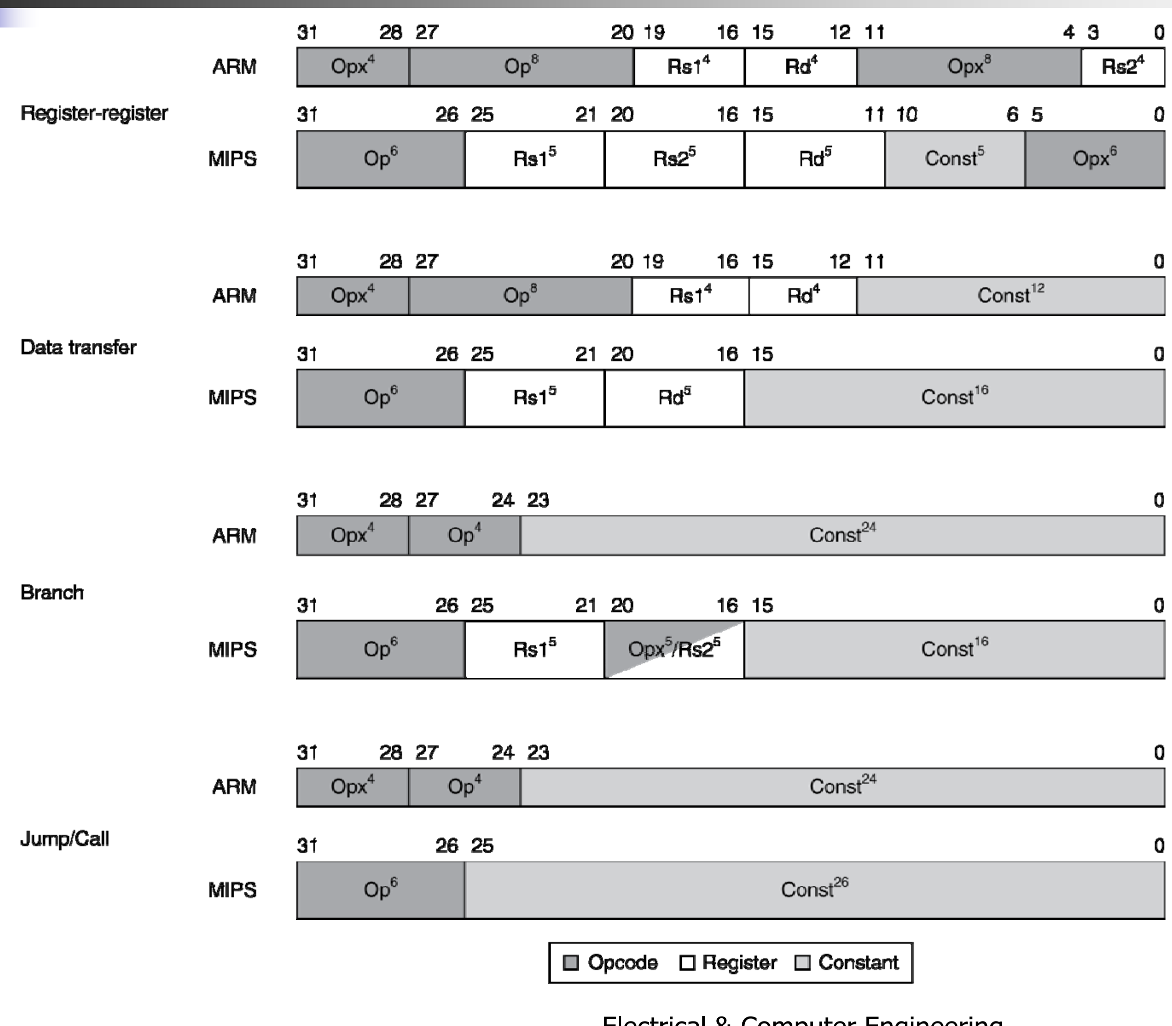
- ARM: the most popular embedded core
- Similar basic set of instructions to MIPS

| | ARM | MIPS |
|---|---|---|
| Date announced | 1985 | 1985 |
| Instruction size | 32 bits | 32 bits |
| Address space | 32-bit flat | 32-bit flat |
| Data alignment | Aligned | Aligned |
| Data addressing modes | 9 | 3 |
| Registers | 15 × 32-bit | 31 × 32-bit |
| Input/output | Memory mapped | Memory mapped |

# Compare and Branch in ARM

- Uses condition codes for result of an arithmetic/logical instruction
  - Negative, zero, carry, overflow
  - Compare instructions to set condition codes without keeping the result
- Each instruction can be conditional
  - Top 4 bits of instruction word: condition value
  - Can avoid branches over single instructions

# Instruction Encoding



# 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

–*“The path toward operation complexity is thus fraught with peril. To avoid these problems, designers have moved toward simpler instructions”*

# Alternative Architectures

- 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 Intel Architecture (IA)

# The Intel x86 ISA

- Evolution with backward compatibility
  - 8080 (1974): 8-bit microprocessor
    - Accumulator, plus 3 index-register pairs
  - 8086 (1978): 16-bit extension to 8080
    - Complex instruction set (CISC)
  - 8087 (1980): floating-point coprocessor
    - Adds FP instructions and register stack
  - 80286 (1982): 24-bit addresses, MMU
    - Segmented memory mapping and protection
  - 80386 (1985): 32-bit extension (now IA-32)
    - Additional addressing modes and operations
    - Paged memory mapping as well as segments

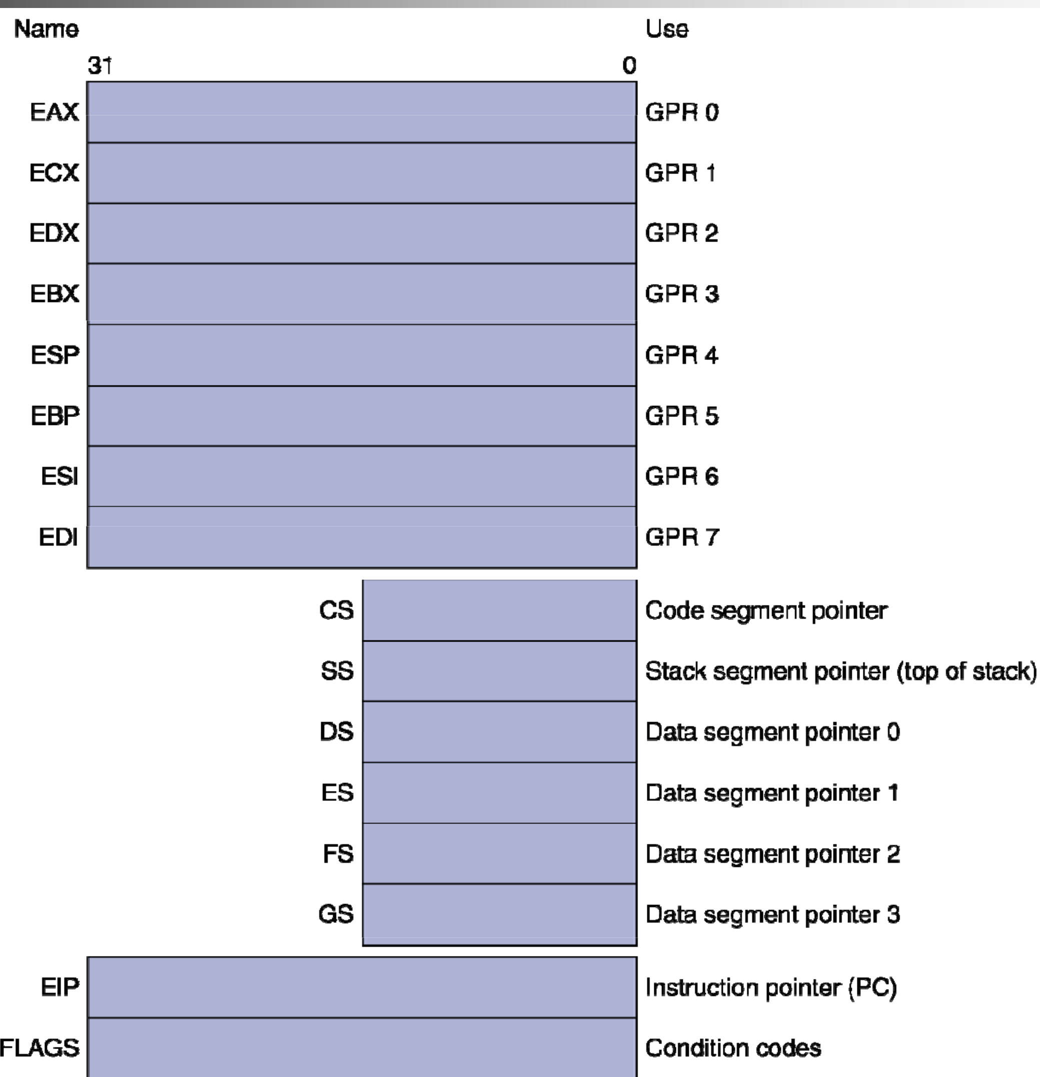
# The Intel x86 ISA

- Further evolution…
  - i486 (1989): pipelined, on-chip caches and FPU
    - Compatible competitors: AMD, Cyrix, …
  - Pentium (1993): superscalar, 64-bit datapath
    - Later versions added MMX (Multi-Media eXtension) instructions
    - The infamous FDIV bug
  - Pentium Pro (1995), Pentium II (1997)
    - New microarchitecture (see Colwell, *The Pentium Chronicles*)
  - Pentium III (1999)
    - Added SSE (Streaming SIMD Extensions) and associated registers
  - Pentium 4 (2001)
    - New microarchitecture
    - Added SSE2 instructions

# The Intel x86 ISA

- And further…
  - AMD64 (2003): extended architecture to 64 bits
  - EM64T – Extended Memory 64 Technology (2004)
    - AMD64 adopted by Intel (with refinements)
    - Added SSE3 instructions
  - Intel Core (2006)
    - Added SSE4 instructions, virtual machine support
  - AMD64 (announced 2007): SSE5 instructions
    - Intel declined to follow, instead…
  - Advanced Vector Extension (announced 2008)
    - Longer SSE registers, more instructions
- If Intel didn't extend with compatibility, its competitors would!
  - Technical elegance ≠ market success

# Basic x86 Registers



# IA-32 Register Restrictions

- Registers are not "general purpose" – note the restrictions below

| Mode | Description | Register restrictions | MIPS equivalent |
|---|---|---|---|
| Register indirect | Address is in a register. | not ESP or EBP | `lw $s0,0($s1)` |
| Based mode with 8- or 32-bit displacement | Address is contents of base register plus displacement. | not ESP or EBP | `lw $s0,100($s1) # ≤16-bit # displacement` |
| Base plus scaled index | The address is Base + ($2^{Scale}$ x Index) where Scale has the value 0, 1, 2, or 3. | Base: any GPR Index: not ESP | `mul $t0,$s2,4` `add $t0,$t0,$s1` `lw $s0,0($t0)` |
| Base plus scaled index with 8- or 32-bit displacement | The address is Base + ($2^{Scale}$ x Index) + displacement where Scale has the value 0, 1, 2, or 3. | Base: any GPR Index: not ESP | `mul $t0,$s2,4` `add $t0,$t0,$s1` `lw $s0,100($t0) # ≤16-bit # displacement` |

**FIGURE 2.42 IA-32 32-bit addressing modes with register restrictions and the equivalent MIPS code.** The Base plus Scaled Index addressing mode, not found in MIPS or the PowerPC, is included to avoid the multiplies by four (scale factor of 2) to turn an index in a register into a byte address (see Figures 2.34 and 2.36). A scale factor of 1 is used for 16-bit data, and a scale factor of 3 for 64-bit data. Scale factor of 0 means the address is not scaled. If the displacement is longer than 16 bits in the second or fourth modes, then the MIPS equivalent mode would need two more instructions: a `lui` to load the upper 16 bits of the displacement and an `add` to sum the upper address with the base register `$s1`. (Intel gives two different names to what is called Based addressing mode—Based and Indexed—but they are essentially identical and we combine them here.)

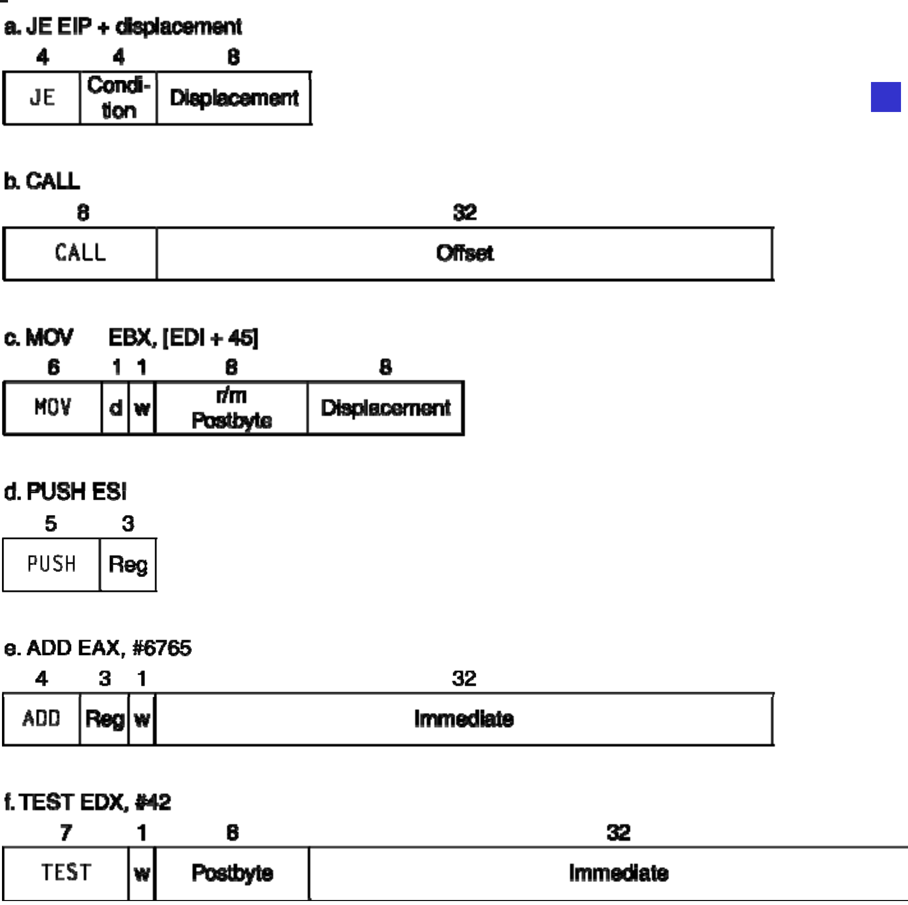
# Basic x86 Addressing Modes

- 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




- Variable length encoding
  - Postfix bytes specify addressing mode
  - Prefix bytes modify operation
    - Operand length, repetition, locking, …

# Implementing IA-32

- Complex instruction set makes implementation difficult
  - Hardware translates instructions to simpler microoperations
    - Simple instructions: 1–1
    - Complex instructions: 1–many
  - Microengine similar to RISC
  - Market share makes this economically viable
- Comparable performance to RISC
  - Compilers avoid complex instructions

# Intel Architecture

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

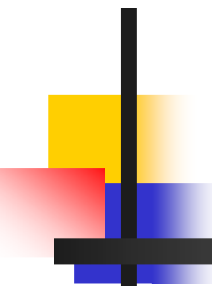
"adding new features as someone might add clothing to a packed bag"

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

# A dominant architecture: 80x86

- 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"*



# 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`"

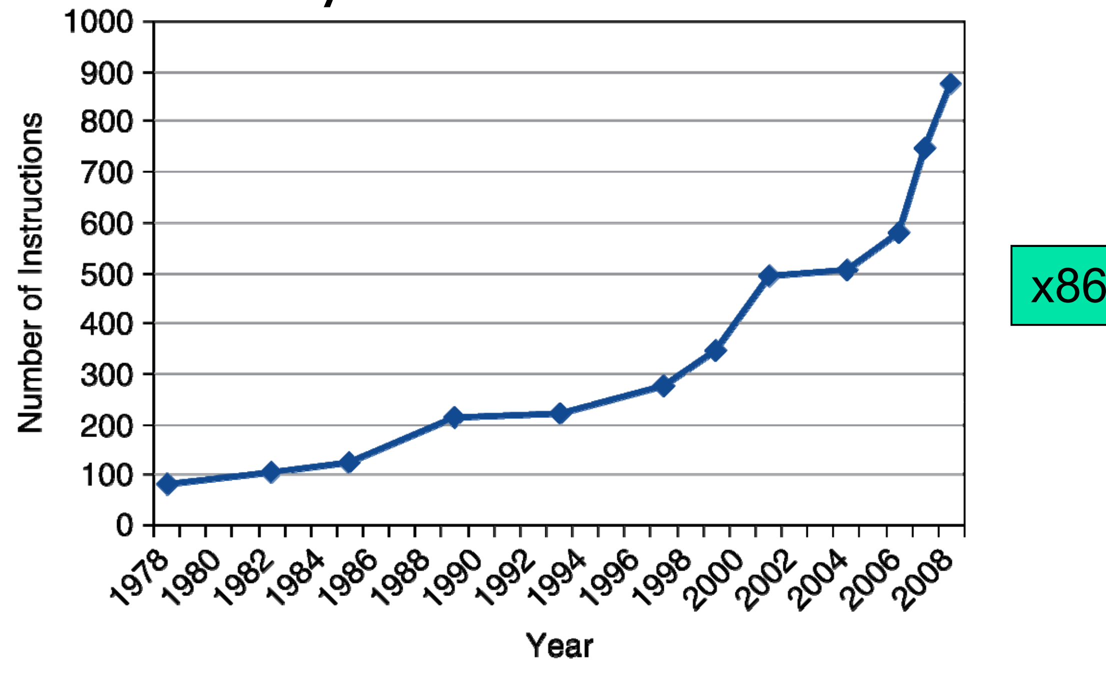
*decrement counter, if not 0 goto loop*

# Fallacies

- Powerful instruction ⇒ higher performance
  - Fewer instructions required
  - But complex instructions are hard to implement
    - 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

- Backward compatibility ⇒ instruction set doesn't change
  - But they do accrete more instructions




x86 instruction set

# Pitfalls

- Sequential words are not at sequential addresses
  - Increment by 4, not by 1!
- Keeping a pointer to an automatic variable after procedure returns
  - e.g., passing pointer back via an argument
  - Pointer becomes invalid when stack popped

# Concluding Remarks

- Design principles
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Make the common case fast
  4. Good design demands good compromises
- Layers of software/hardware
  - Compiler, assembler, hardware
- MIPS: typical of RISC ISAs
  - x86

# Concluding Remarks

- Measure MIPS instruction executions in benchmark programs
  - Consider making the common case fast

| Instruction class | MIPS examples | SPEC2006 Int | SPEC2006 FP |
|---|---|---|---|
| Arithmetic | add, sub, addi | 16% | 48% |
| Data transfer | lw, sw, lb, lbu, lh, lhu, sb, lui | 35% | 36% |
| Logical | and, or, nor, andi, ori, sll, srl | 12% | 4% |
| Cond. Branch | beq, bne, slt, slti, sltiu | 34% | 8% |
| Jump | j, jr, jal | 2% | 0% |

# 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 number | | |
| J | op | 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 `$t4` ≠ `$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 number** |

| | | |
|---|---|---|
| **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
beq $t4,$t5,Label
```

Next instruction is at Label if $\$t4 \neq \$t5$
Next instruction is at Label if $\$t4=\$t5$

- Formats:

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

- 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**

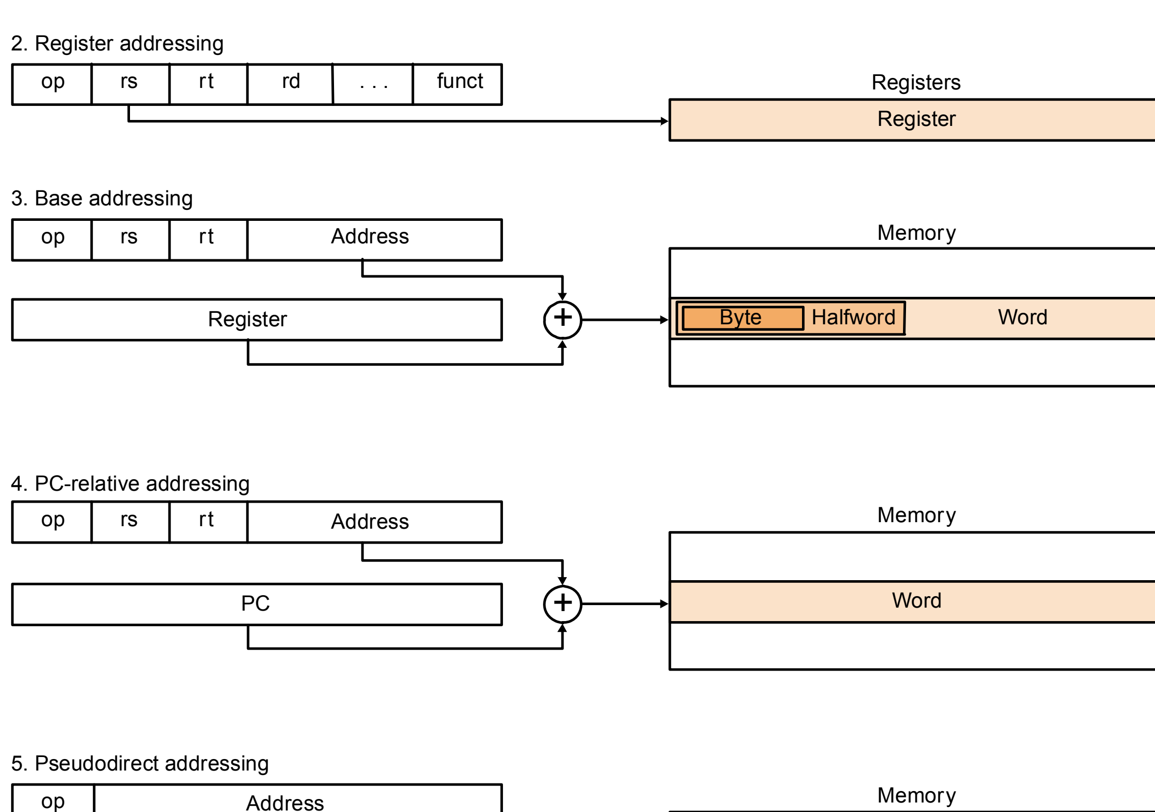
| Name | Example | Comments |
|---|---|---|
| 32 registers | $s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at | Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle 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**

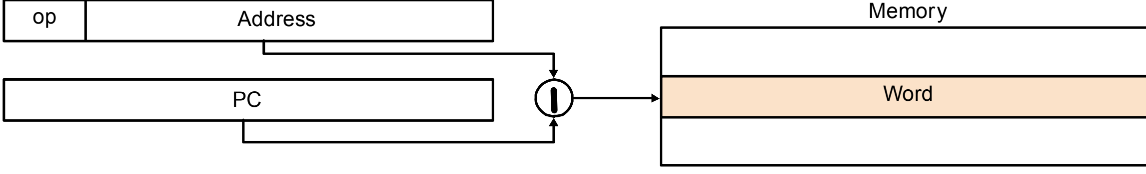
| Category | Instruction | Example | Meaning | Comments |
|---|---|---|---|---|
| Arithmetic | add | add $s1, $s2, $s3 | $s1 = $s2 + $s3 | Three operands; data in registers |
| | subtract | sub $s1, $s2, $s3 | $s1 = $s2 - $s3 | Three operands; data in registers |
| | add immediate | addi $s1, $s2, 100 | $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 |

1. Immediate addressing

2. Register addressing




5. Pseudodirect addressing



# Summary

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