

---

## Addressing in branches and jumps

- Since most of the branches are close to the current instruction, we can use \$pc as the register to be added, leading to a range of  $\pm 2^{15}$  from the current value in \$pc
- This is called PC-relative addressing
- Distance or range of branch can be stretched by using the fact that all MIPS instructions are 4 bytes long
- Jump-and-link instructions may go anywhere in the process and hence, they are performed with J-type instructions
- The 26-bit field also uses word addressing, allowing for a jump that is  $2^{28}$  bytes
- The full 32-bit addressing can be achieved by using a jump register instruction

---

# Addressing in branches and jumps

- Branching far away (done automatically by assembler by inverting the condition)

beq \$s0, \$s1, L1

gets translated to

bne \$s0, \$s1, L2

j L1

L2:

---

# MIPS addressing mode summary

- Register addressing: operand is a register
- Base or displacement addressing: the operand is at the memory location whose address is the sum of a register and a constant in the instruction
- Immediate addressing: the operand is a constant within the instruction itself
- PC-relative addressing: the address is the sum of the PC and a constant in the instruction
- Pseudodirect addressing: the jump address is the 26 bits of the instruction concatenated with the upper bits of the PC

# MIPS addressing mode summary

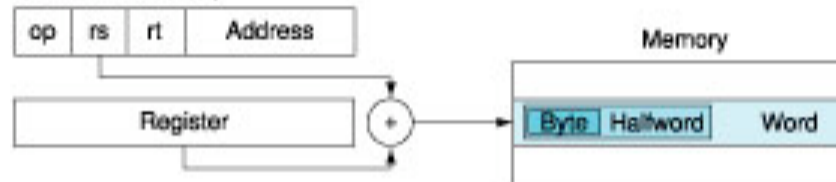
## 1. Immediate addressing



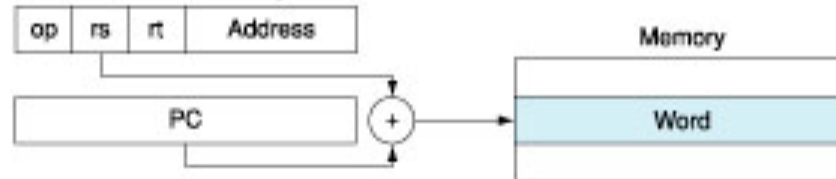
## 2. Register addressing



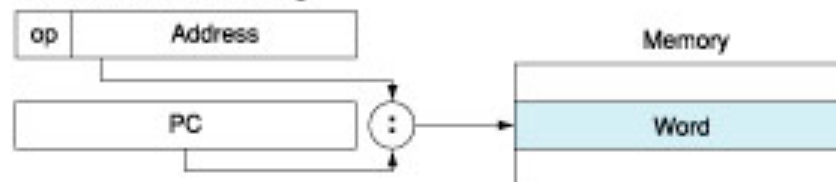
## 3. Base addressing



## 4. PC-relative addressing

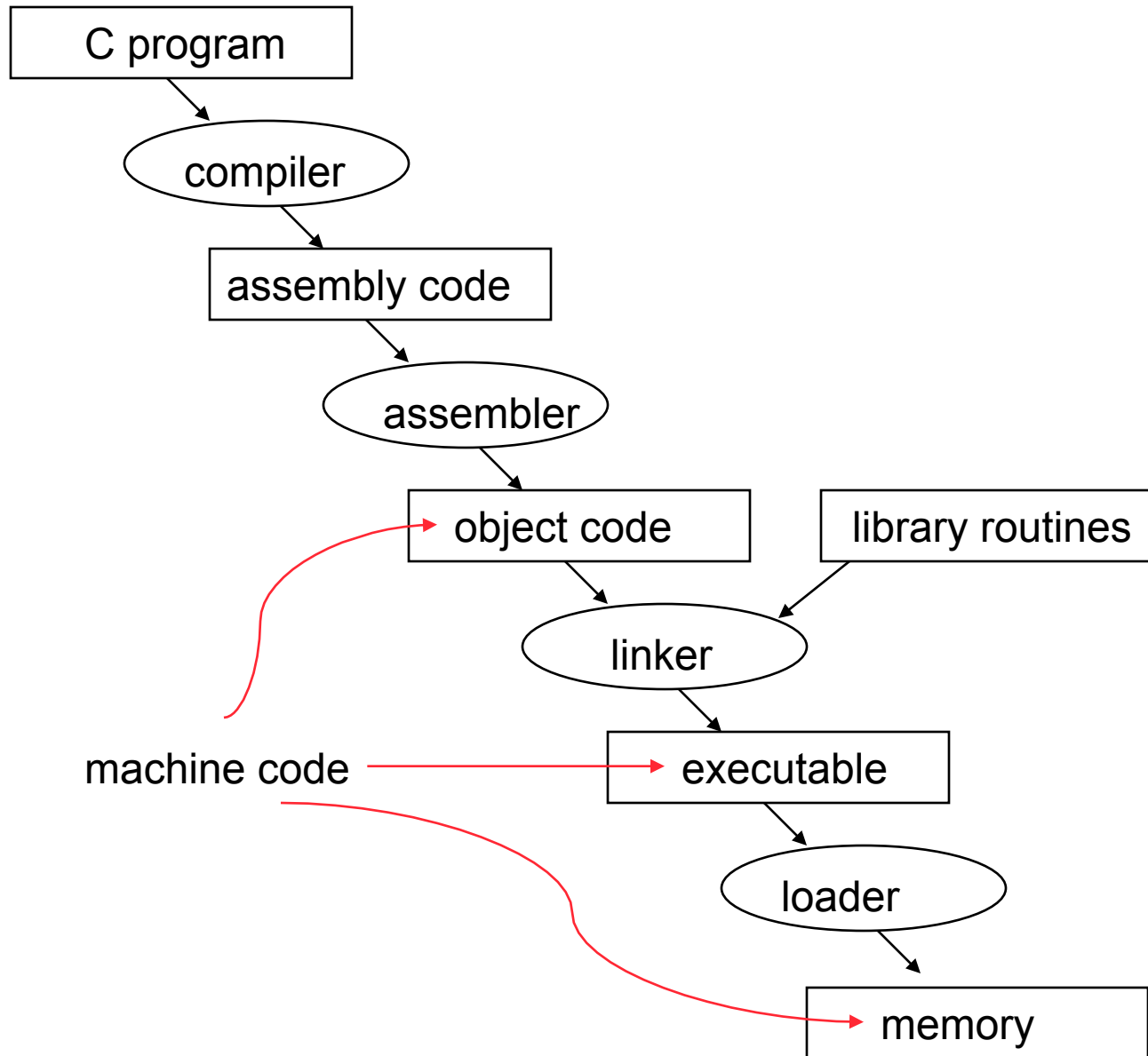


## 5. Pseudodirect addressing



---

# The Code Translation Hierarchy



---

# Compiler

- Transforms the C program into an assembly language program
- Advantages of high-level languages
  - many fewer lines of code
  - easier to understand and debug
- Today's optimizing compilers can produce assembly code nearly as good as an assembly language programming expert and often better for large programs
  - good – smaller code size, faster execution
  - and even lower power consuming!

---

# Assembler

- Transforms symbolic assembler code into object (machine) code
- Advantages of assembler
  - much easier than remembering instruction binary codes
  - can use labels for addresses – and let the assembler do the arithmetic
  - can use pseudo-instructions
    - e.g., “move \$t0, \$t1” exists only in assembler (would be implemented using “add \$t0,\$t1,\$zero”)
- However, must remember that machine language is the underlying reality
- And, when considering performance, you should count **real** instructions **executed**, not code size

---

## Other Tasks of the Assembler

- Determines binary addresses corresponding to all labels
  - keeps track of labels used in branches and data transfer instructions in a **symbol table**
    - pairs of symbols and addresses
- Converts pseudo-instructions to legal assembly code
  - register \$at is reserved for the assembler to do this
- Converts branches to far away locations into a branch followed by a jump
- Converts instructions with large immediates into a load upper immediate followed by an or immediate
- Converts numbers specified in decimal and hexadecimal into their binary equivalents
- Converts characters into their ASCII equivalents

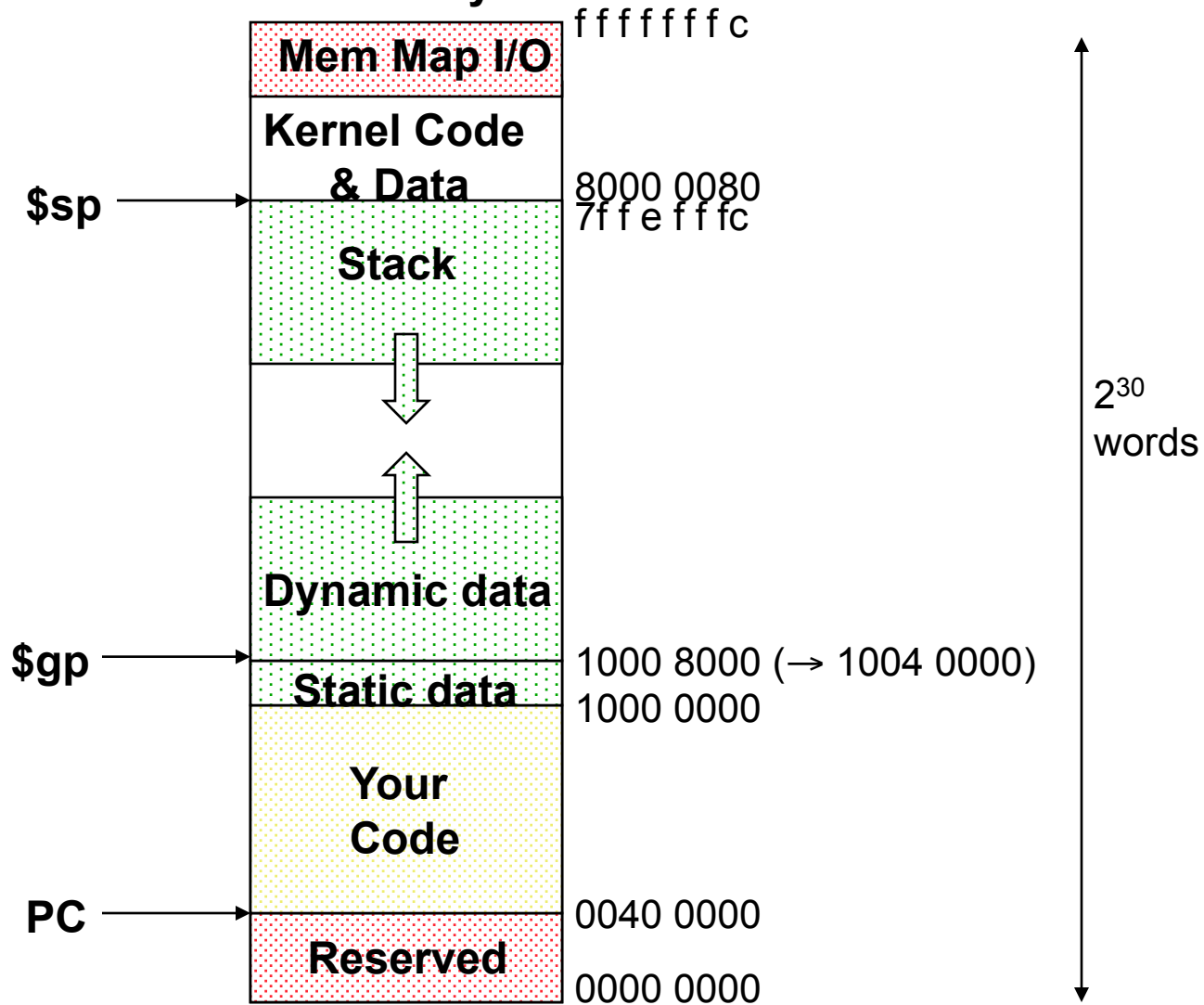
---

# Typical Object File Pieces

Object file header	Text segment	Data segment	Relocation information	Symbol table	Debugging information
--------------------	--------------	--------------	------------------------	--------------	-----------------------

- Object file header: size and position of following pieces
- Text module: assembled object (machine) code
- Data module: data accompanying the code
  - static data - allocated throughout the program
  - dynamic data - grows and shrinks as needed by the program
- Relocation information: identifies instructions (data) that use (are located at) **absolute addresses** – those that are not relative to a register (e.g., jump destination addr) – when the code and data is loaded into memory
- Symbol table: remaining undefined labels (e.g., external references)
- Debugging information

# MIPS (spim) Memory Allocation



---

# Linker

- Takes all of the independently assembled code segments and “stitches” (links) them together
  - Much faster to patch code and recompile and reassemble that patched routine, than it is to recompile and reassemble the entire program
- Decides on memory allocation pattern for the code and data modules of each segment
  - remember, segments were assembled in isolation so **each** assumes its code’s starting location is 0x0040 0000 and its static data starting location is 0x1000 0000
- Absolute addresses must be **relocated** to reflect the new starting location of each code and data module
- Uses the symbol table information to resolve all remaining undefined labels
  - branches, jumps, and data addresses to external segments

# Example

<b>Object file header</b>			
	<b>Name</b>	<b>Procedure A</b>	
	<b>Text Size</b>	<b>0x100</b>	
	<b>Data size</b>	<b>0x20</b>	
<b>Text Segment</b>	<b>Address</b>	<b>Instruction</b>	
	0	lw \$a0, 0(\$gp)	
	4	jal 0	
	...	...	
<b>Data segment</b>	0	(X)	
	...	...	
<b>Relocation information</b>	<b>Address</b>	<b>Instruction Type</b>	<b>Dependency</b>
	0	lw	X
	4	jal	B
<b>Symbol Table</b>	<b>Label</b>	<b>Address</b>	
	X	-	
	B	-	

# Example

<b>Object file header</b>			
	<b>Name</b>	<b>Procedure B</b>	
	<b>Text Size</b>	<b>0x200</b>	
	<b>Data size</b>	<b>0x30</b>	
<b>Text Segment</b>	<b>Address</b>	<b>Instruction</b>	
	0	sw \$a1, 0(\$gp)	
	4	jal 0	
	...	...	
<b>Data segment</b>	0	(Y)	
	...	...	
<b>Relocation information</b>	<b>Address</b>	<b>Instruction Type</b>	<b>Dependency</b>
	0	lw	Y
	4	jal	A
<b>Symbol Table</b>	<b>Label</b>	<b>Address</b>	
	Y	-	
	A	-	

---

# Example

<b>Executable file header</b>		
	<b>Text size</b>	0x300
	<b>Data size</b>	0x50
<b>Text segment</b>	<b>Address</b>	<b>Instruction</b>
	0x0040 0000	Lw \$a0, 0x8000(\$gp)
	0x0040 0004	Jal 0x0040 0100
	...	...
	0x0040 0100	Sw \$a1, 0x8020(\$sp)
	0x0040 0104	Jal 0x0040 0000
	...	...
<b>Data segment</b>	<b>Address</b>	
	0x1000 0000	(x)
	...	...
	0x1000 0020	(Y)
	...	...

---

# Loader

- Loads (copies) the executable code now stored on disk into memory at the starting address specified by the **operating system**
- Initializes the machine registers and sets the stack pointer to the first free location (0x7ffe fffc)
- Copies the parameters (if any) to the main routine onto the stack
- Jumps to a start-up routine (at PC addr 0x0040 0000 on xspim) that copies the parameters into the argument registers and then calls the main routine of the program with a `jal main`

---

# Dynamically Linked Libraries

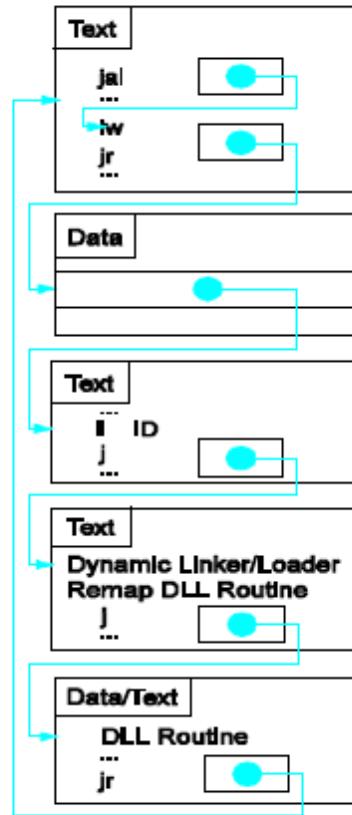
- Static linked libraries have some disadvantages
  - libraries become part of the code, program has to be recompiled in order to use newer versions of the libraries
  - the whole library is loaded even if all the routines in the library are not used
    - Standard C library is 2.5 MB
- Dynamically linked libraries (DLLs) – library routines are not linked and loaded until the program is run
  - lazy procedure linkage approach: a procedure is linked only after it is called
  - extra overhead (for dynamic linking) the first time a routine is called + extra space overhead for the information needed for dynamic linking, but no overhead on subsequent calls

---

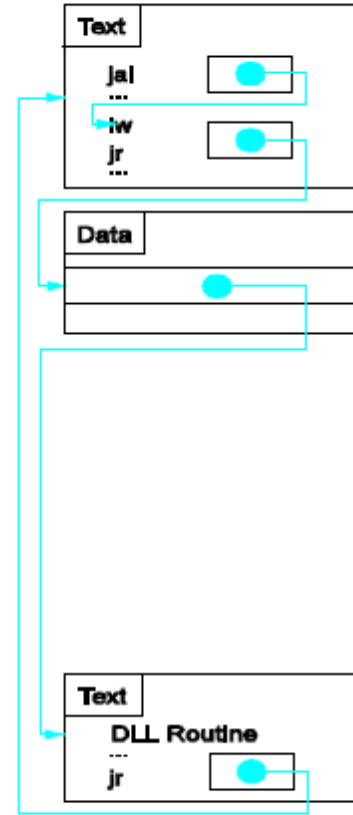
# Dynamically Linked Libraries

- Lazy procedure linking: the **routine** is linked only after it is called
- This uses a level of indirection
- There is a **dummy routine** for each external procedure
- The dummy routine puts the ID of the desired library routine into a register
- The dummy routine then calls (jumps to) a dynamic linker-loader **routine**
- The linker-loader finds the desired routine and remaps it
- The linker-loader also change the indirect jump to point directly to the **routine**
- So further calls does not access the linker-loader

# Dynamically Linked Libraries



(a) First call to DLL routine



(b) Subsequent calls to DLL routine

---

## Full Example – Sort in C

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

- Allocate registers to program variables
- Produce code for the program body
- Preserve registers across procedure invocations

---

## Full Example – Sort in C

- Register allocation: \$a0 and \$a1 for the two arguments, \$t0 for the temp variable – no need for saves and restores as we're not using \$s0-\$s7 and this is a leaf procedure (won't need to re-use \$a0 and \$a1)

```
swap:      sll $t1, $a1, 2
           add $t1, $a0, $t1
           lw $t0, 0($t1)
           lw $t2, 4($t1)
           sw $t2, 0($t1)
           sw $t0, 4($t1)
           jr $ra
```

---

## Full Example – Sort in C

```
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);
        }
    }
}
```

- Register allocation: arguments v and n use \$a0 and \$a1, i and j use \$s0 and \$s1
- Must save \$a0, \$a1, and \$ra before calling the leaf procedure

---

## Full Example – Sort in C

- The outer for loop looks like this: (note the use of pseudo-instrs)

```
                move $s0, $zero           # initialize the loop
for1tst:        slt $t0, $s0, $a1,
                beq $t0, $zero,exit1
                ... body of inner loop ...
                addi $s0, $s0, 1
                j for1tst

exit1:
```

---

## Full Example – Sort in C

- The inner for loop looks like this:

```
                                addi $s1, $s0, -1           # initialize the loop
for2tst:                        slti $t0,$s1, 0
                                bne $t0,$zero, exit2
                                sll $t1, $s1, 2
                                add $t2, $a0, $t1
                                lw $t3, 0($t2)
                                lw $t4, 4($t2)
                                slt $t0, $t4, $t3
                                beq $t0, $zero, exit2
                                ... body of inner loop ...
                                addi $s1, $s1, -1
                                j for2tst
exit2:
```

---

## Full Example – Sort in C

- Since we repeatedly call “swap” with \$a0 and \$a1, we begin “sort” by copying its arguments into \$s2 and \$s3 – must update the rest of the code in “sort” to use \$s2 and \$s3 instead of \$a0 and \$a1
- Must save \$ra at the start of “sort” because it will get overwritten when we call “swap”
- Must also save \$s0-\$s3 so we don’t overwrite something that belongs to the procedure that called “sort”

---

## Full Example – Sort in C

```
sort:      addi $sp, $sp, -20
           sw $ra, 16($sp)
           sw $s3, 12($sp)
           sw $s2, 8($sp)
           sw $s1, 4($sp)
           sw $s0, 0($sp)
           move $s2, $a0
           move $s3, $a1
           ...
           move $a0, $s2      # the inner loop body starts here
           move $a1, $s1
           jal swap
           ...
exit1:    lw $s0, 0($sp)
           ...
           addi $sp, $sp, 20
           jr $ra
```

---

### Procedure body

```
swap: sll    $t1, $a1, 2           # reg $t1 = k * 4
      add    $t1, $a0, $t1        # reg $t1 = v + (k * 4)
                                           # reg $t1 has the address of v[k]
      lw     $t0, 0($t1)          # reg $t0 (temp) = v[k]
      lw     $t2, 4($t1)          # reg $t2 = v[k + 1]
                                           # refers to next element of v
      sw     $t2, 0($t1)          # v[k] = reg $t2
      sw     $t0, 4($t1)          # v[k+1] = reg $t0 (temp)
```

### Procedure return

```
jr     $ra                        # return to calling routine
```

### Saving registers

```

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

Move parameters	<pre> move   \$s2,\$a0 # copy parameter \$a0 into \$s2 (save \$a0) move   \$s3,\$a1 # copy parameter \$a1 into \$s3 (save \$a1)         </pre>
Outer loop	<pre> move   \$s0,\$zero# i = 0 for1tst: slt\$t0,\$s0,\$s3# reg \$t0 = 0 if \$s0 &lt; \$s3 (i &lt; n) beq    \$t0,\$zero,exit1# go to exit1 if \$s0 &lt; \$s3 (i &lt; n)         </pre>
Inner loop	<pre> addi   \$s1,\$s0,-1# j = i - 1 for2tst: slti\$t0,\$s1,0    # reg \$t0 = 1 if \$s1 &lt; 0 (j &lt; 0) bne    \$t0,\$zero,exit2# go to exit2 if \$s1 &lt; 0 (j &lt; 0) sll    \$t1,\$s1,2# reg \$t1 = j * 4 add    \$t2,\$s2,\$t1# reg \$t2 = v + (j * 4) lw     \$t3,0(\$t2)# reg \$t3 = v[j] lw     \$t4,4(\$t2)# reg \$t4 = v[j + 1] slt    \$t0,\$t4,\$t3 # reg \$t0 = 0 if \$t4 &lt; \$t3 beq    \$t0,\$zero,exit2# go to exit2 if \$t4 &lt; \$t3         </pre>
Pass parameters and call	<pre> move   \$a0,\$s2      # 1st parameter of swap is v (old \$a0) move   \$a1,\$s1     # 2nd parameter of swap is j jal    swap        # swap code shown in Figure 2.25         </pre>
Inner loop	<pre> addi   \$s1,\$s1,-1# j -- 1 j      for2tst     # jump to test of inner loop         </pre>
Outer loop	<pre> exit2: addi   \$s0,\$s0,1    # i += 1 j      for1tst     # jump to test of outer loop         </pre>

### Restoring registers

```

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
    
```

### Procedure return

```

jr     $ra        # return to calling routine
    
```

---

## Full Example – Sort in C

Gcc optimization	Relative performance	Cycles	Instruction count	CPI
none	1.00	159B	115B	1.38
O1	2.37	67B	37B	1.79
O2	2.38	67B	40B	1.66
O3	2.41	66B	45B	1.46

---

# Arrays versus Pointers

- C and MIPS versions of two procedures to clear a sequence of words in memory

```
Clear1 ( int array [ ] , int size)
{
    int i ;
    for ( i = 0; i < size; i ++ )
        array [ i ] = 0;
}
```

```
Clear2 ( int *array , int size)
{
    int *p, i=0 ;
    for ( p=&array[0]; p<&array[size]; p ++ )
        *p = 0;
}
```

---

# Array Version

```
Clear1 ( int array [ ] , int size)
{
    int i ;
    for ( i = 0; i < size; i ++)
        array [ i ] = 0;
}
```

- array is in \$a0 and size in \$a1 and i is allocated to \$t0

```

        move    $t0, $zero           # i = 0
Loop1:  sll     $t1, $t0, 2           # $t1 = 4 * i
        add     $t2, $a0, $t1       # $t2 = addr. of array
        sw     $zero, 0($t2)        # array[i] = 0
        addi   $t0, $t0, 1          # i ++
        slt    $t3, $t0, $a1 #check end of loop (i<size)
        bne    $t3, $zero, Loop1    # if i < size goto Loop1
```

---

## Pointer version

```
Clear2 ( int *array , int size)
{
    int *p, i=0 ;
    for(p=&array[0]; p<&array[size]; p ++)
        *p = 0;
}
```

- array is in \$a0 and size in \$a1 and p is allocated to \$t0

```
        move    $t0, $a0          # p = addr. of array[0]
        sll     $t1, $a1, 2        # $t1 = 4 * size
        add     $t2, $a0, $t1     #$t2 = addr. of array[size]
Loop2:  sw      $zero, 0($t0)     # store 0 in *p
        addi   $t0, $t0, 4        # p = p + 4
        slt    $t3, $t0, $t2     # $t3=(p<&array[size])
        bne    $t3, $zero, Loop2 #if p < last addr. Go Loop2
```

---

# Comparison

- The pointer version reduces the # of instructions per iteration from 6 to 4
- Many optimizing compilers will generate this code, even for array-based C code

---

# History of the Intel 80x86

- IA-32 is the product of several independent groups evolved over 20 years
- 1975: 8080 introduced
  - 8-bit microprocessor
  - Accumulator machine (a single register for arithmetic operations)
- 1978: 8086 introduced
  - 16 bit microprocessor
  - Accumulator plus dedicated registers
- 1980: IBM selects 8088 as basis for IBM PC
  - 8088 is 8-bit external bus version of 8086
- 1980: 8087 floating point coprocessor
  - adds 60 floating point instructions
  - uses hybrid stack/register scheme

---

# History of the Intel 80x86

- 1982: 80286 introduced
  - 24-bit address
  - memory mapping & protection
- 1985: 80386 introduced
  - 32-bit address
  - 32-bit general purpose registers
- 1989: 80486 introduced
- 1992: Pentium introduced
- 1995: Pentium Pro introduced
- 1996: Pentium with MMX (multimedia extensions)
  - 57 new instructions
  - Primarily for multimedia applications
- 1997: Pentium II (Pentium Pro with MMX)

---

# History of the Intel 80x86

- 1999: Pentium III Introduced
  - Supports Intel's Streaming SIMD extension (SSE)
    - Additional multimedia instructions
    - Four 32-bit floating point operations in parallel
- 2001: Intel introduces SSE2
  - 144 new instructions
  - Pairs of 64 bit floating point operations in parallel
- 2003: AMD enhances IA-32 to 64 bits
  - all registers are widen to 64 bits
  - number of registers are increased to 16 and number of 128-bit SSE registers to 16
- 2004: Intel embraces AMD64 relabeling it EM64T

---

# History of the Intel 80x86

- Intel architecture was due to the desire for backward compatibility
  - Highly irregular architecture
  - Over 50 million sold per year
- Intel's 80x86s architectures are far less regular and far more complex to understand and program than MIPS

---

# Intel 80x86 Integer Registers

Name	31	0	Use
EAX	[Register]		GPR 0
ECX	[Register]		GPR 1
EDX	[Register]		GPR 2
EBX	[Register]		GPR 3
ESP	[Register]		GPR 4
EBP	[Register]		GPR 5
ESI	[Register]		GPR 6
EDI	[Register]		GPR 7
	CS	[Register]	Code segment pointer
	SS	[Register]	Stack segment pointer (top of stack)
	DS	[Register]	Data segment pointer 0
	ES	[Register]	Data segment pointer 1
	FS	[Register]	Data segment pointer 2
	GS	[Register]	Data segment pointer 3
EIP	[Register]		Instruction pointer (PC)
EFLAGS	[Register]		Condition codes

---

# X86 Operand Types

- IA-32 instructions typically have two operands, where one operand is both a source and a destination operand.
- Possible combinations include

<u>Source/destination type</u>	<u>Second source type</u>
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

- No memory-memory or immediate-immediate
- Immediates can be 8, 16, or 32 bits

## The x86 offers several different addressing modes for accessing memory

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 <sup>Scale</sup> × 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 <sup>Scale</sup> × 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.)

---

# 80x86 Integer Instructions

- Data movement (move, push, pop)
- Arithmetic and logic (logic ops, tests CCs, shifts, integer and decimal arithmetic)
- Control flow (branches, jumps, calls, returns)
- String instructions (move and compare)

Instruction	Function
JE name	if equal(condition code) {EIP=name}; EIP-128 ≤ name < EIP+128
JMP name	EIP=name
CALL name	SP=SP-4; M[SP]=EIP+5; EIP=name;
MOVW EBX, [EDI+45]	EBX=M[EDI+45]
PUSH ESI	SP=SP-4; M[SP]=ESI
POP EDI	EDI=M[SP]; SP=SP+4
ADD EAX, #6765	EAX= EAX+6765
TEST EDX, #42	Set condition code (flags) with EDX and 42
MOVSL	M[EDI]=M[ESI]; EDI=EDI+4; ESI=ESI+4

**FIGURE 2.43** Some typical IA-32 instructions and their functions. A list of frequent operations appears in Figure 2.44. The CALL saves the EIP of the next instruction on the stack. (EIP is the Intel PC.)

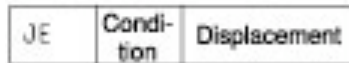
---

# 80x86 Instruction Format

- Instructions sizes vary from 1 to 17 bytes

a. JE EIP + displacement

4 4 8



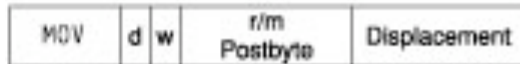
b. CALL

8 32



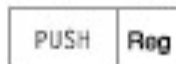
c. MOV EBX, [EDI + 45]

6 1 1 8 8



d. PUSH ESI

5 3



e. ADD EAX, #6765

4 3 1 32



f. TEST EDX, #42

7 1 8 32

