

Chapter 2

Computer Abstractions and Technology

Lesson 5: MIPS (cont...)

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 (naïve):
 - 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

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

◆ 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

32-bit Constants

- ◆ Most constants are small
 - 16-bit immediate is sufficient
- ◆ For the occasional 32-bit constant

`lui rt, constant`

- Copies 16-bit constant to left 16 bits of `rt`
- Clears right 16 bits of `rt` to 0

`lui $s0, 61`

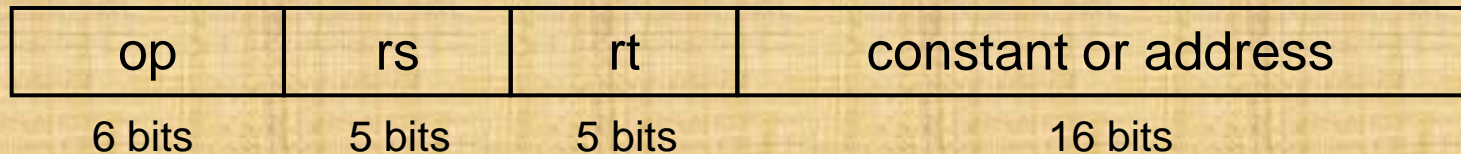
0000 0000 0111 1101	0000 0000 0000 0000
---------------------	---------------------

`ori $s0, $s0, 2304`

0000 0000 0111 1101	0000 1001 0000 0000
---------------------	---------------------

Branch Addressing

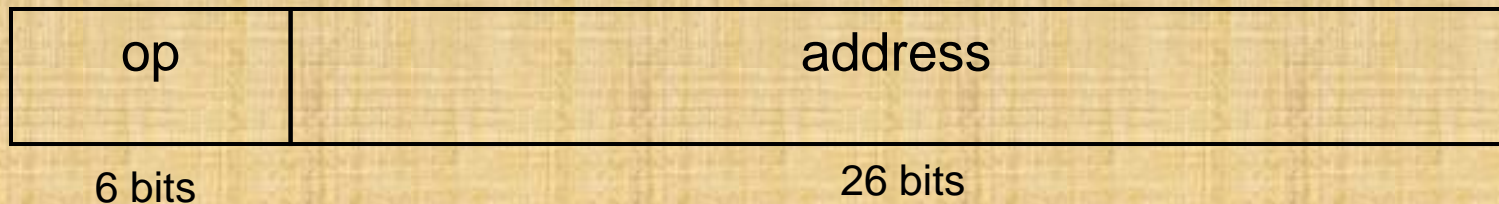
- ◆ Branch instructions specify
 - Opcode, two registers, target address
- ◆ Most branch targets are near branch
 - Forward or backward



- PC-relative addressing
 - Target address = $PC + \text{offset} \times 4$
 - Shift Left the PC by 2 first, then add offset

Jump Addressing

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



- (Pseudo)Direct jump addressing
 - Target address = $PC_{31..28} : (\text{address} \times 4)$
 - When jump address is 26 bits of the instruction concatenated with the upper bits of the PC

Target Addressing Example

- ◆ Loop code from earlier example
 - Assume Loop at location 80000

Loop: sll \$t1, \$s3, 2	80000	0	0	19	9	4	0
add \$t1, \$t1, \$s6	80004	0	9	22	9	0	32
lw \$t0, 0(\$t1)	80008	35	9	8	0		
bne \$t0, \$s5, Exit	80012	5	8	21	2		
addi \$s3, \$s3, 1	80016	8	19	19	1		
j Loop	80020	2	20000				
Exit: ...	80024						

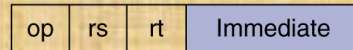
Branching Far Away

- ◆ 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
L2:     j    L1
        ...
```

Addressing Mode Summary

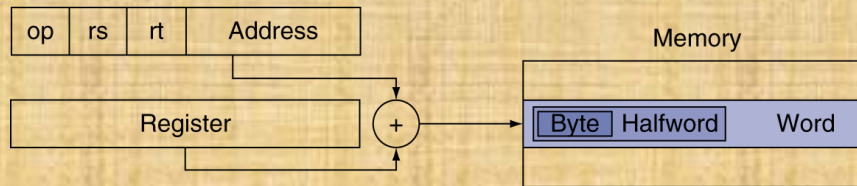
1. Immediate addressing



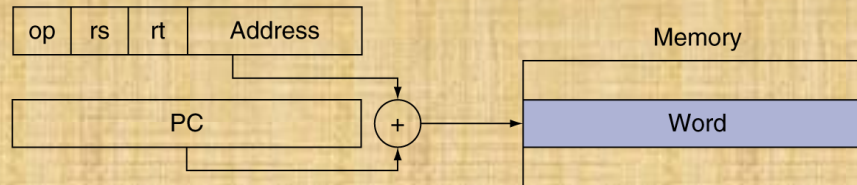
2. Register addressing



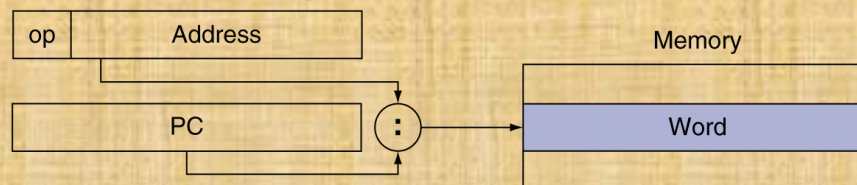
3. Base addressing



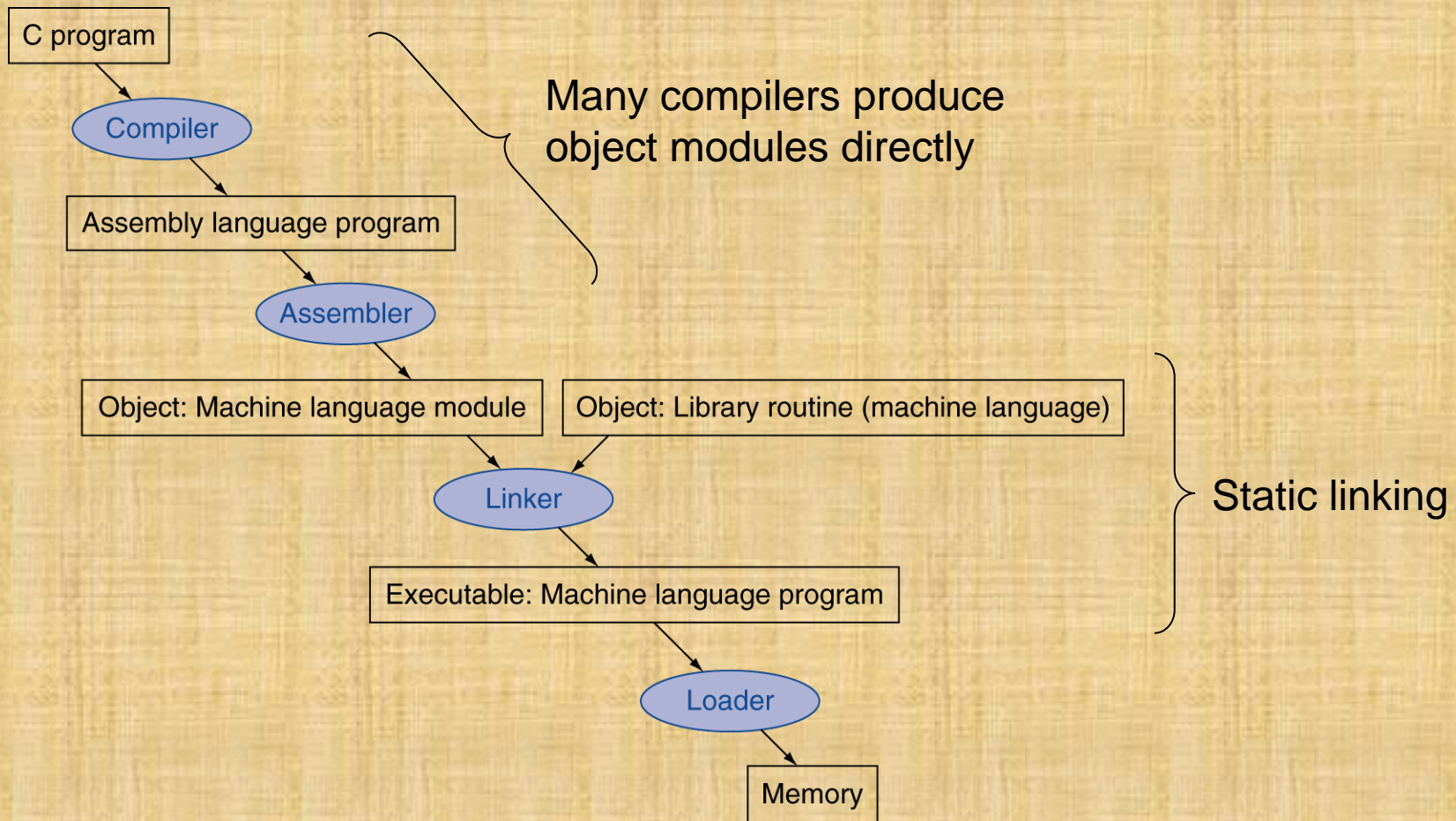
4. PC-relative addressing



5. Pseudodirect addressing



Translation and Startup



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

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 params
	move \$s3, \$a1	# save \$a1 into \$s3	
for1tst:	move \$s0, \$zero	# i = 0	Outer loop
	slt \$t0, \$s0, \$s3	# \$t0 = 0 if \$s0 ≥ \$s3 (i ≥ n)	
	beq \$t0, \$zero, exit1	# go to exit1 if \$s0 ≥ \$s3 (i ≥ n)	
for2tst:	addi \$s1, \$s0, -1	# j = i - 1	
	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	Inner loop
	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)	Pass params & call
	move \$a1, \$s1	# 2nd param of swap is j	
	jal swap	# call swap procedure	
	addi \$s1, \$s1, -1	# j -= 1	Inner loop
	j for2tst	# jump to test of inner loop	
exit2:	addi \$s0, \$s0, 1	# i += 1	Outer loop
	j for1tst	# jump to test of 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
```

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

Example: Clearing and Array

```
clear1(int array[], int size) {  
    int i;  
    for (i = 0; i < size; i += 1)  
        array[i] = 0;  
}
```

```
        move $t0,$zero    # i = 0  
loop1: sll $t1,$t0,2      # $t1 = i * 4  
        add $t2,$a0,$t1  # $t2 =  
                        # &array[i]  
        sw $zero, 0($t2) # array[i] = 0  
        addi $t0,$t0,1   # i = i + 1  
        slt $t3,$t0,$a1 # $t3 =  
                        # (i < size)  
        bne $t3,$zero,loop1 # if (...)  
                        # goto loop1
```

```
clear2(int *array, int size) {  
    int *p;  
    for (p = &array[0]; p < &array[size];  
        p = p + 1)  
        *p = 0;  
}
```

```
        move $t0,$a0     # p = & array[0]  
        sll $t1,$a1,2    # $t1 = size * 4  
        add $t2,$a0,$t1  # $t2 =  
                        # &array[size]  
loop2: sw $zero,0($t0)  # Memory[p] = 0  
        addi $t0,$t0,4   # p = p + 4  
        slt $t3,$t0,$t2 # $t3 =  
                        #(p<&array[size])  
        bne $t3,$zero,loop2 # if (...)  
                        # goto loop2
```

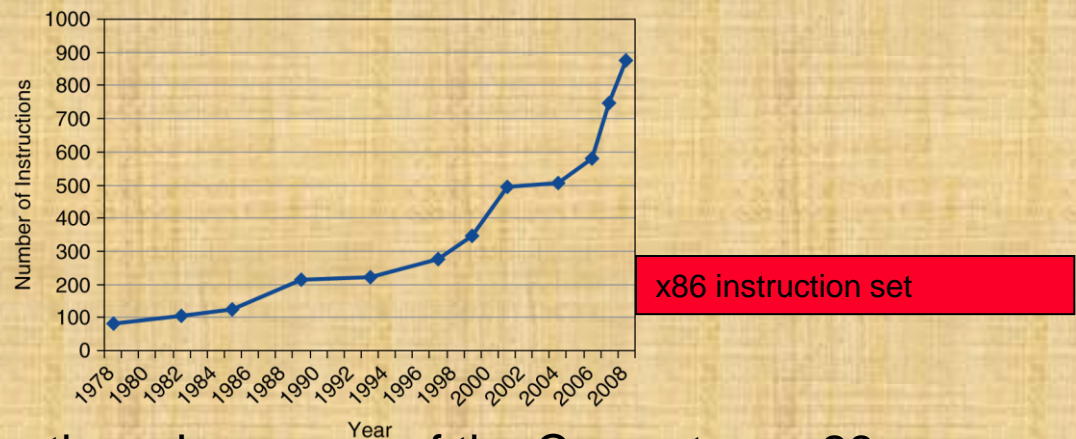
Fallacies

- ◆ Powerful instruction \Rightarrow 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 \Rightarrow more errors and less productivity

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
- ◆ Backward compatibility \Rightarrow instruction set doesn't change
 - But they do accrete more instructions



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

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%

Next Time

- ◆ We start Ch 3 – Arithmetic for Computers
- ◆ **Assignment:**
- ◆ An HW has been issued (see on the class web page)
- ◆ Due by next Tu Sept 29
 - 2 additional points for any early returning day
 - up to 8 additional points for those who return it by R Sept 24
- ◆ On Sept 28 we have a Review class to prepare for the Exam (Oct 1)