Computer Organization and Structure

Bing-Yu Chen
National Taiwan University

Instructions: Language of the Computer

- Operations and Operands
  - of the Computer Hardware
- Signed and Unsigned Numbers
- Representing Instructions
  - in the Computer
- Logical Operations
- Instructions for Making Decisions
- Supporting Procedures
  - in Computer Hardware
- Communicating with People
- MIPS Addressing
  - for 32-Bit Immediates and Addresses
- Translating and Starting a Program
- Arrays vs. Pointers

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

Arithmetic Operations

- Add and Subtract, 3 operands
  - 2 sources and 1 destination
- operand order is fixed
  - destination first
  - all arithmetic operations have this form

- Example:
  - C code: a = b + c
  - MIPS code: add a, b, c

Arithmetic Operations

- *Design Principle 1*:
  - simplicity favors regularity
    - Regularity makes implementation simpler
    - Simplicity enables higher performance at lower cost

Arithmetic Examples

- compiling two C assignments into MIPS
  - C code:
    ```
    a = b + c;
    d = a - e;
    ```
  - MIPS code:
    ```
    add a, b, c
    sub d, a, e
    ```

- compiling a complex C assignment into MIPS
  - C code:
    ```
    f = (g + h) – (i + j)
    ```
  - MIPS code:
    ```
    add $t0, g, h       # temp t0 = g + h
    add $t1, i, j       # temp t1 = i + j
    sub f, $t0, $t1     # f = t0 - t1
    ```

Register Operands

- Of course this complicates some things...
  - C code: a = b + c + d;
  - MIPS code:
    ```
    add a, b, c
    add a, a, d
    ```
  - where a & b & c & d mean **registers**

- Arithmetic instructions use register operands
  - operands must be **registers**

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
    - c.f. main memory: millions of locations

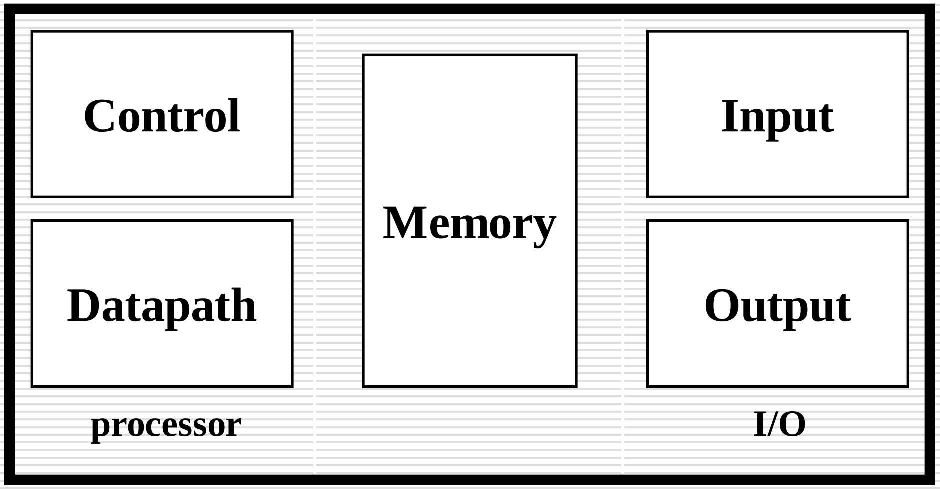
Register Operand Example

- C code: f = (g + h) – (i + j)
  - assume f, …, j in $s0, …, $s4
- MIPS code:

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

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 Operands

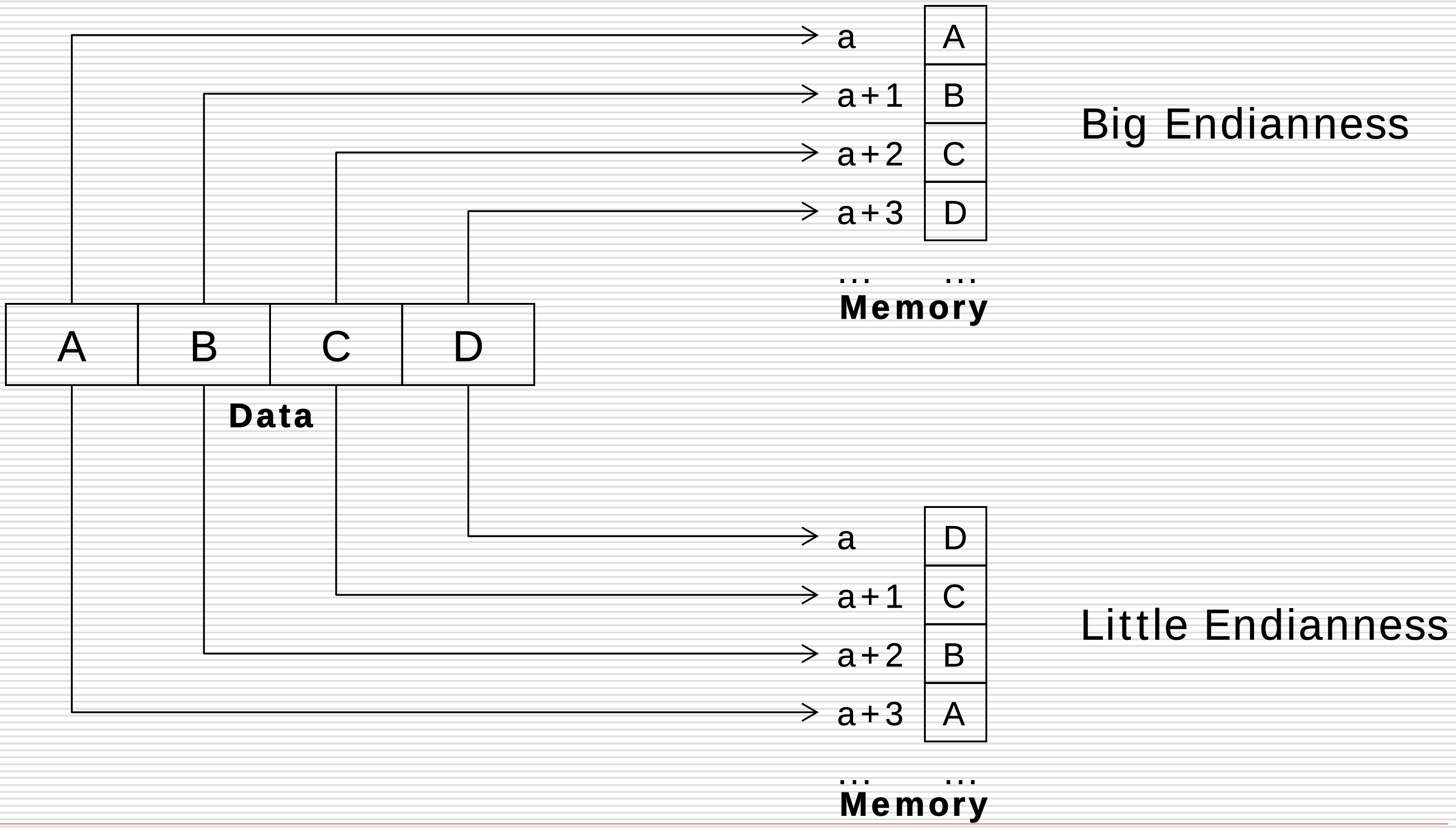
- Main memory used for composite data
  - Arrays, structures, dynamic data
- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory
- MIPS is Big Endian
  - Most-Significant Byte at least address of a word
  - c.f. Little Endian: Least-Significant Byte at least address

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 8-bit byte of memory

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
…	…

Big Endian vs. Little Endian



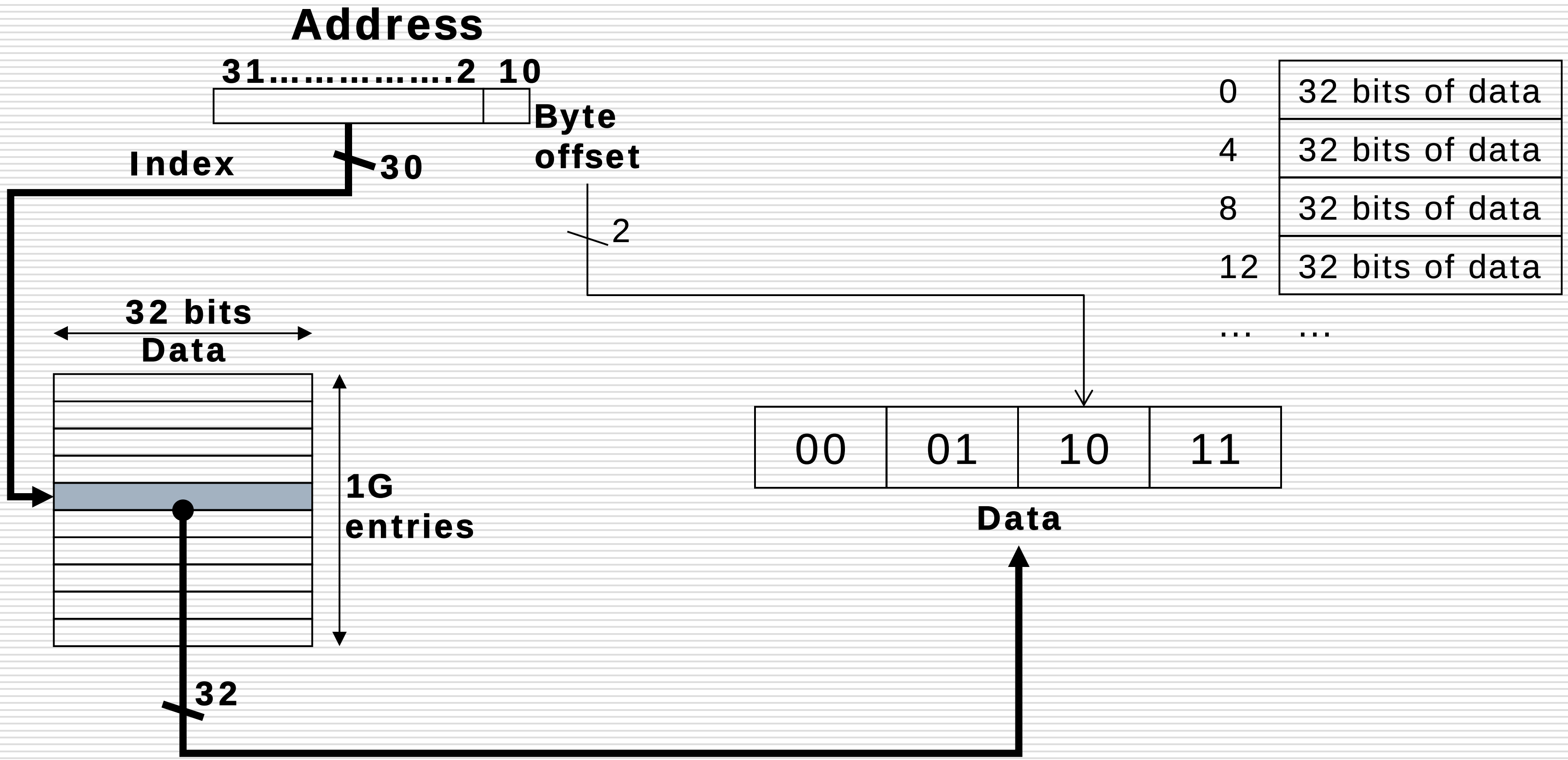
Memory Organization

- “bytes” are nice, but most data items use larger "words"
- for MIPS
  - a word is 32 bits or 4 bytes

0	32 bits of data
4	32 bits of data
8	32 bits of data
12	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 (*alignment restriction*)
  - Address must be a multiple of 4
  - What are the least 2 significant bits of a word address?

Word Addressing



Load & Store Instructions

- C code: g = h + A[8];
  - g in $s1, h in $s2, ***base address*** of A in $s3

- MIPS code:
  ```
  lw   $t0, 32($s3)
  add $s1, $s2, $t0
  ```
  - index 8 requires ***offset*** of 32
    - 4 bytes per word

- can refer to registers by name (e.g., $s2, $t0) instead of number

Load & Store Instructions

- C code: A[12] = h + A[8];
  - h in $s2, base address of A in $s3
- MIPS code:

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

- store word has *destination last*
- remember arithmetic operands are registers, not memory
  - can't write: add 48($s3), $s2, 32($s3)

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!

Constants

- Small constants are used quite frequently
  - 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?

The Constant Zero

- MIPS register 0 ($zero) is the constant 0
  - Cannot be overwritten
- Useful for common operations
  - add $t2, $s1, $zero
    - e.g., move between registers

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

*e.g. subi

Numbers

- Bits are just bits (no inherent meaning)
  - conventions define relationship between bits and numbers
- Binary numbers (base 2)
  - decimal: $0 \ldots 2^n-1$
- Of course it gets more complicated:
  - numbers are finite (overflow)
  - fractions and real numbers
  - negative numbers

Unsigned Binary Integers

- Given an n-bit number

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

- 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

2's-Complement Signed Integers

- Given an n-bit number

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

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

2's-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 2's-complement representation
- Some specific numbers
  - 0: 0000 0000 … 0000
  - –1: 1111 1111 … 1111
  - Most-negative: 1000 0000 … 0000
  - Most-positive: 0111 1111 … 1111

Signed Negation

- Complement and add 1
  - Complement means 1 → 0, 0 → 1

$$x + \overline{x} = 1111...111_2 = -1$$

$$\overline{x} + 1 = -x$$

- Example: negate +2
  - +2 = 0000 0000 … 0010_2
  - –2 = 1111 1111 … 1101_2 + 1
    = 1111 1111 … 1110_2

- “negate” and “complement” are quite different!

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

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 instruction encoding@Fig.2.19@P.135
MIPS register conventions@Fig.2.14@P.121

MIPS R-format Instructions

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

- op = operation code (opcode)
  - basic operation of the instruction
- rs / rt / rd
  - register source / destination operand
- 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

- $00000010001100100100000000100000_2 = 02324020_{16}$

Hexadecimal

- Base 16
  - Compact representation of bit strings
  - 4 bits per hex digit

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

- Example: eca8 6420
  - 1110 1100 1010 1000 0110 0100 0010 0000

MIPS I-format Instructions

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

- Immediate arithmetic and load/store instructions
  - rs / rt: source or destination 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

I-format Example

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

- lw $t0, 32($s2)

lw	$s2	$t0	32
35	18	8	32
100011	10010	01000	0000000000100000

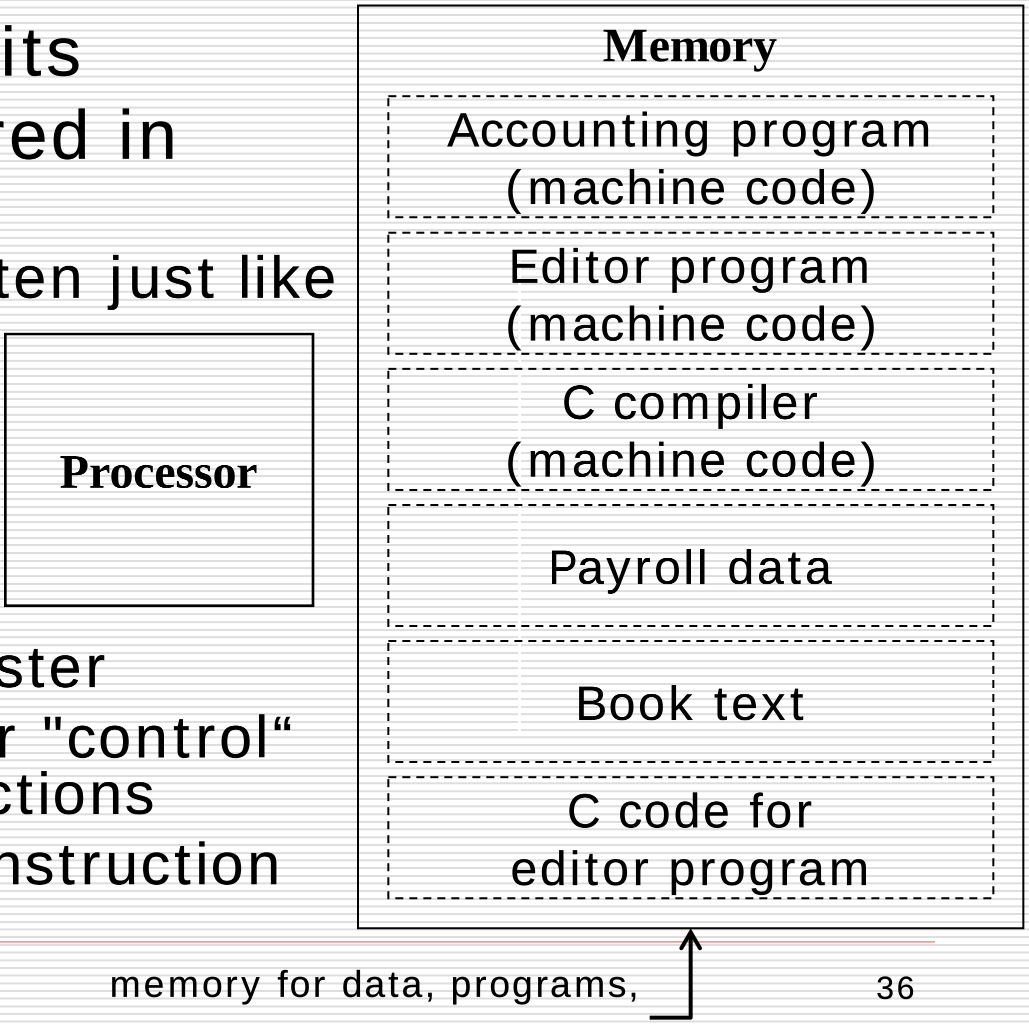
C / MIPS / Machine Languages

- C: A[300] = h + A[300]
- MIPS:
```
lw   $t0, 1200($t1)
add  $t0, $s2, $t0
sw   $t0, 1200($t1)
```
- Machine Language:

35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

Stored Program Concept

- Instructions are bits
- Programs are stored in memory
  - to be read or written just like data
- 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


Processor
Memory
Accounting program (machine code)
Editor program (machine code)
C compiler (machine code)
Payroll data
Book text
C code for editor program

memory for data, programs, compilers, editors, etc.

Logical Operations

Logical operations	C operators	MIPS instructions
Shift left	<<	sll
Shift right	>>	srl
Bitwise AND	&	and, andi
Bitwise OR	|	or, ori

Logical Operations

Instruction	Example
and	and $s1, $s2, $s3
or	or $s1, $s2, $s3
nor	nor $s1, $s2, $s3
and immediate	andi $s1, $s2, 100
or immediate	ori $s1, $s2, 100
shift left logical	sll $s1, $s2, 10
shift right logical	srl $s1, $s2, 10

- There is no NOT, since …
  - NOT (A) = NOT (A OR 0) = A NOR 0

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)

Shift Operations

- NOTICE
  - shift left/right logical is not *I-type*

- Example: sll $t2, $s0, 4
- Machine Language:

op	rs	rt	rd	shamt	funct
special	**none**	**$s0**	**$t2**	**4**	**sll**
0	**0**	**16**	**10**	**4**	**0**

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

Conditional Operations

- 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: ....
```

Unconditional Operations

- MIPS **unconditional branch** instructions:
  - j Label
- (Un-)Conditional Branch Example:

```
if (i==j)                 bne $s3, $s4, Else
    f=g+h;                add $s0, $s1, $s2
else                      j Exit
    f=g-h;          Else: sub $s0, $s1, $s2
                    Exit: ...
```

Assembler calculates addresses

- Can you build a simple **for** / **while** loop ?

While Loop

C:

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

- assume i in $s3, k in $s5, address of save in $s6

MIPS:

```
Loop:  sll    $t1, $s3, 2        # $t1=4*i
       add    $t1, $t1, $s6      # $t1=addr. of save[i]
       lw     $t0, 0($t1)        # $t0=save[i]
       bne    $t0, $s5, Exit     # go to Exit if save[i]!=k
       addi   $s3, $s3, 1        # i+=1
       j      Loop               # go to Loop
Exit:
```

Control Flow

- set on less than:

```
if ($s3 < $s4)              slti $t1, $s3, $s4
    $t1=1;
else
    $t1=0;
```

- can use this instruction to build
  "blt $s1, $s2, Label"
  - can now build general control structures
- NOTE
  - the assembler needs a register to do this,
  - there are policy of use conventions for registers

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

Case/Switch in C

```
switch (k) {
    case 0:        f=i+j;      break;      /* k=0 */
    case 1:        f=g+h;      break;      /* k=1 */
    case 2:        f=g-h;      break;      /* k=2 */
    case 3:        f=i-j;      break;      /* k=3 */
}
```

Case/Switch in MIPS -- using Jump Address Table

```
        slt     $t3, $s5, $zero         # test if k < 0
        bne     $t3, $zero, Exit        # if k < 0, go to Exit
        slti    $t3, $s5, 4             # test if k < 4
        beq     $t3, $zero, Exit        # if k >= 4, go to Exit
        sll     $t1, $s5, 2             # $t1 = 4 * k
        add     $t1, $t1, $t4           # $t1=addr. of JumpTable[k]
        lw      $t0, 0($t1)             # $t0= JumpTable[k]
        jr      $t0                     # jump based on $t0
L0:     add     $s0, $s3, $s4           # k = 0, so f gets i + j
        j       Exit                    # end of this case so go to Exit
L1:     add     $s0, $s1, $s2           # k = 1, so f gets g + h
        j       Exit                    # end of this case so go to Exit
L2:     sub     $s0, $s1, $s2           # k = 2, so f gets g - h
        j       Exit                    # end of this case so go to Exit
L3:     sub     $s0, $s3, $s4           # k = 3, so f gets i - j
Exit:                                   # end of switch statement
```

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
  - Place parameters in registers
  - Transfer control to procedure
  - Acquire storage for procedure
  - Perform procedure’s operations
  - Place result in register for caller
  - Return to place of call

Register Usage

Name	Register No.	Usage
$zero	0	the constant value 0
$v0-$v1	2-3	values for results & expression evaluation
$a0-$a3	4-7	arguments
$t0-$t7	8-15	temporaries (can be overwritten by callee)
$s0-$s7	16-23	saved (must be saved/restored by callee)
$t8-$t9	24-25	more temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address

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

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 in C

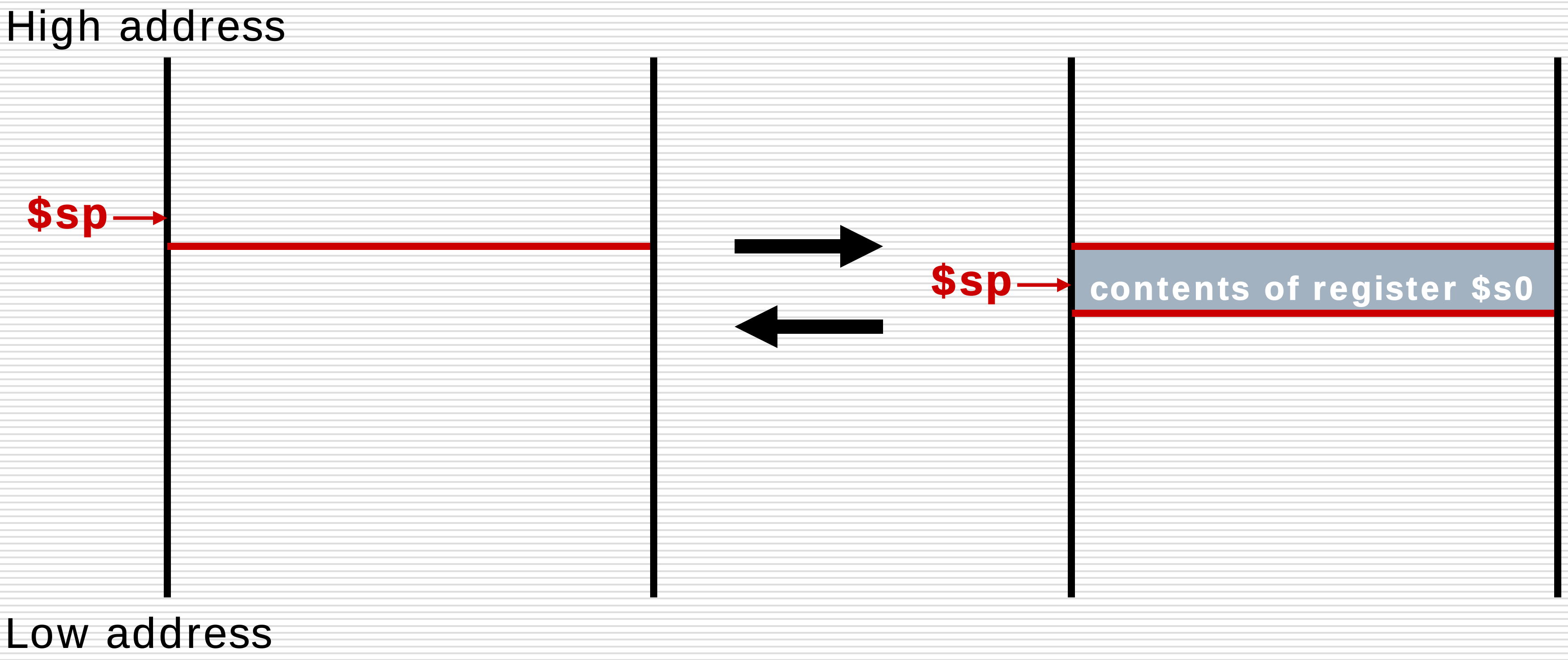
```
int leaf_example (int g, int h, int i, int j) {
    int f;

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

- Assume
  - Arguments g, ..., j in $a0, ..., $a3
  - f in $s0 (hence, need to save $s0 on stack)
  - Result in $v0

Leaf Procedure in MISP

```
addi  $sp, $sp, -4         # adjust stack for saving $s0
sw    $s0, 0($sp)
-------------------------------------------------------------
add   $t0, $a0, $a1        # g+h
add   $t1, $a2, $a3        # i+j
sub   $s0, $t0, $t1        # (g+h)-(i+j)
add   $v0, $s0, $zero      # return f ($v0=$s0+0)
-------------------------------------------------------------
lw    $s0, 0($sp)
addi  $sp, $sp, 4          # adjust stack again
-------------------------------------------------------------
jr    $ra                  # jump back to calling routine
```

The Stack



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

Recursive Procedure in C

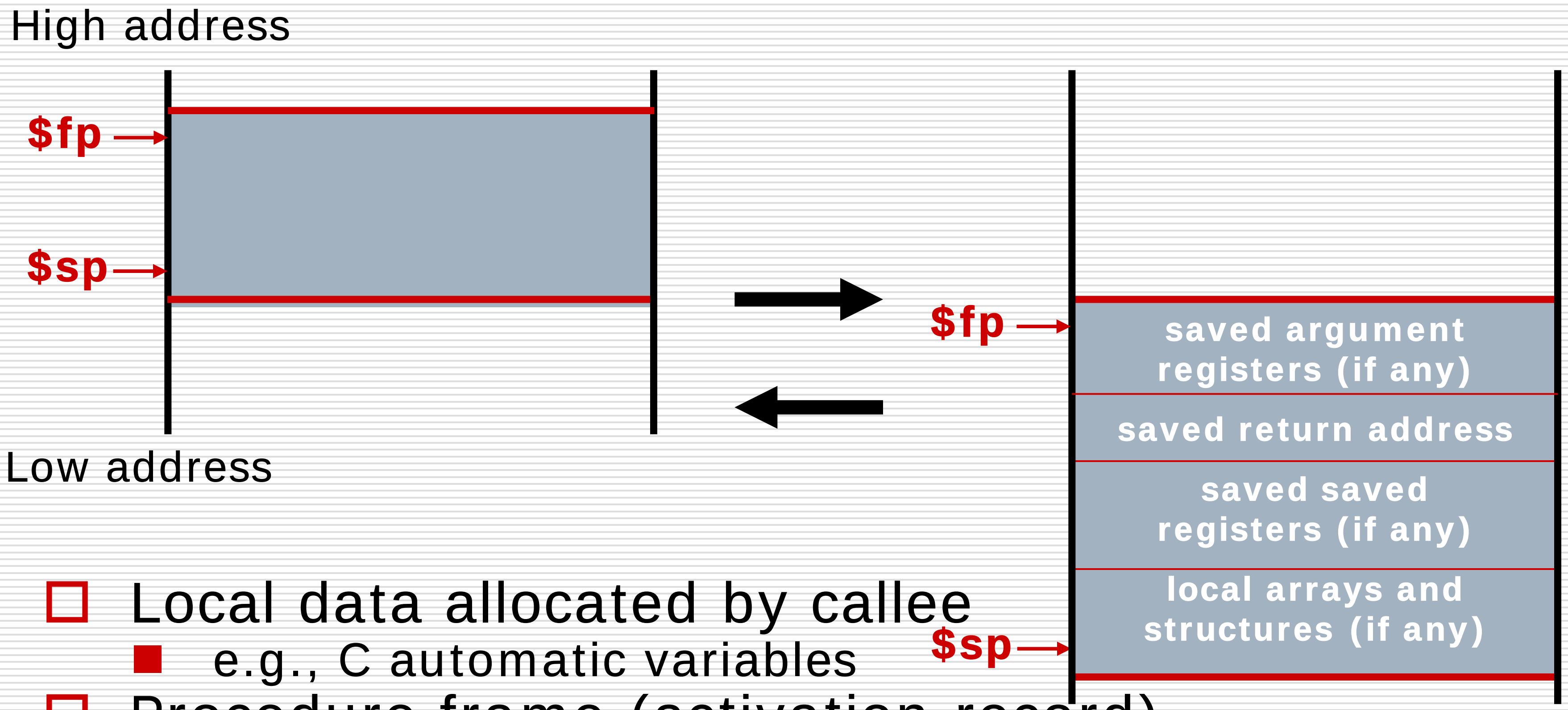
```
int fact (int n) {
    if (n < 1)
        return 1;
    else
        return (n * fact (n - 1));
}
```

- Assume
  - Argument n in $a0
  - Result in $v0

Recursive Procedure in MISP

```
fact:
      addi      $sp, $sp, -8      # adjust stack for 2 items
      sw        $ra, 4($sp)       # save the return address
      sw        $a0, 0($sp)       # save the argument n
      slti      $t0, $a0, 1       # test for n < 1
      beq       $t0, $zero, L1    # if n >= 1, go to L1
      addi      $sp, $sp, 8       # pop 2 items off stack
      addi      $v0, $zero, 1     # return 1
      jr        $ra               # return to after jal
L1:   addi      $a0, $a0, -1      # n >= 1: argument gets (n - 1)
      jal       fact              # call fact with (n - 1)
      lw        $a0, 0($sp)       # return from jal: restore argument n
      lw        $ra, 4($sp)       # restore the return address
      addi      $sp, $sp, 8       # adjust stack pointer to pop 2 items
      mul       $v0, $a0, $v0     # return n * fact (n - 1)
      jr        $ra               # return to the caller
```

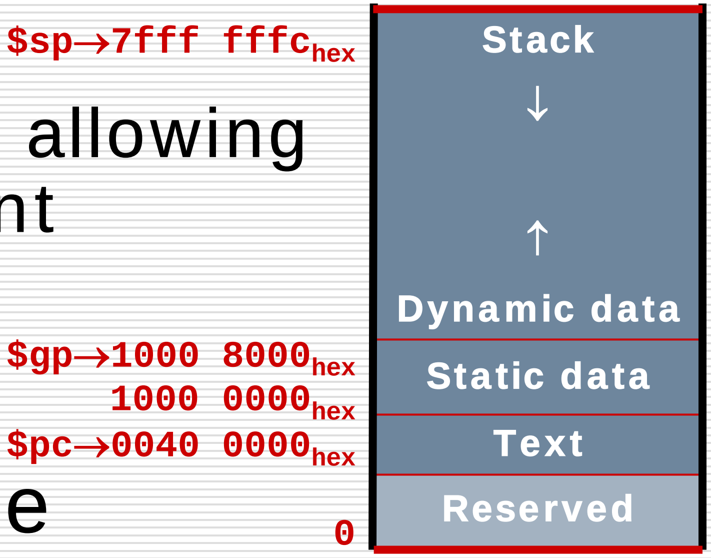
Local Data on the Stack


High address
$fp
$sp
Low address
$fp
$sp
saved argument registers (if any)
saved return address
saved saved registers (if any)
local arrays and structures (if any)

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

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
- Stack: automatic storage



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 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 Procedure in C

```
void strcpy (char x[], char y []) {
    int i;

    i = 0;
    while (x[i] = y[i] != '¥0') {
        i = i + 1;
    }
}
```

- Assume
  - Null-terminated string
  - Addresses of x, y in $a0, $a1, i in $s0

String Copy Procedure in MIPS

```
     addi   $sp, $sp, -4
     sw     $s0, 0($sp)
     add    $s0, $zero, $zero    # i = 0
L1:  add    $t1, $s0, $a1        # address of y[i] in $t1
     lb     $t2, 0($t1)          # $t2 = y[i]
     add    $t3, $s0, $a0        # address of x[i] in $t3
     sb     $t2, 0($t3)          # x[i] = y[i]
     beq    $t2, $zero, L2       # if y[i] == 0, go to L2
     addi   $s0, $s0, 1          # i = i + 1
     j      L1                   # go to L1
L2:  lw     $s0, 0($sp)          # restore old $s0
     addi   $sp, $sp, 4
     jr     $ra
```

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

- Instructions:
  - bne $s0,$s1,L1
  - beq $s0,$s1,L2
- Formats:

I	**op**	**rs**	**rt**	**16 bit number**

- Most branch targets are near branch
  - Forward or backward
- PC-relative addressing
  - Target address = PC + offset × 4
  - PC already incremented by 4 by this time

Jump Addressing

- Instructions:
  - j L1
  - jal L2
- Formats:

J	**op**	**26 bit number**

- Jump targets could be anywhere in text segment
  - Encode full address in instruction
- (Pseudo)Direct jump addressing
  - Target address = $PC_{31\ldots28}$: (address $\times$ 4)

Target Addressing Example

C:

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

MIPS:

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	...					

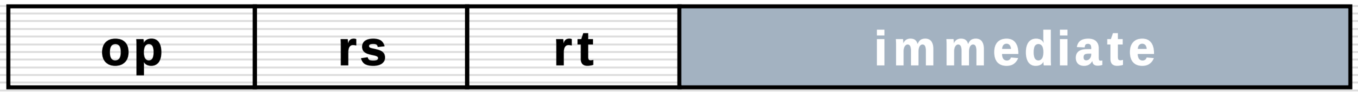
Branching Far Away

- If branch target is too far to encode with 16-bit offset, assembler rewrites the code
- Example

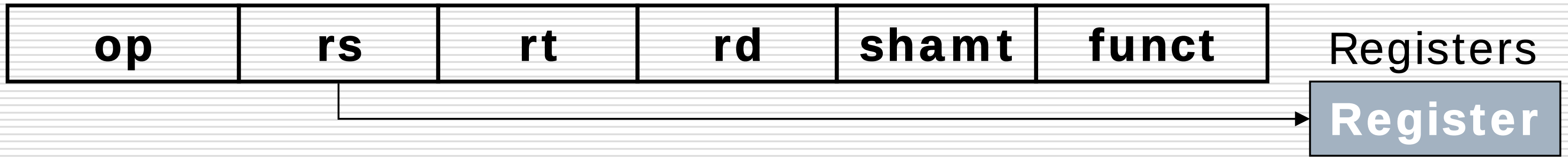
```
            beq $s0,$s1, L1
                  ↓
            bne $s0,$s1, L2
            j L1
L2:         ...
```

Addressing Modes

- Immediate addressing

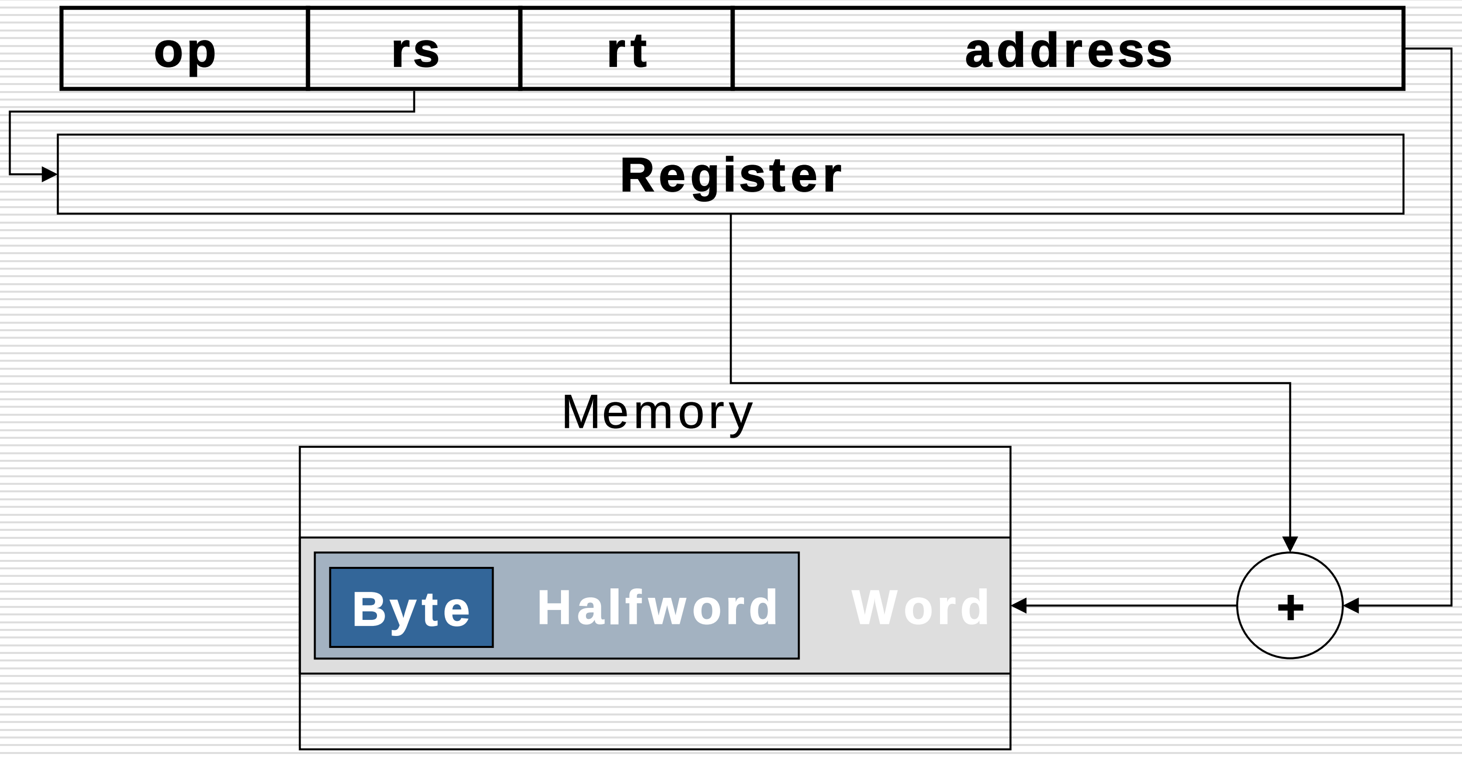


- Register addressing



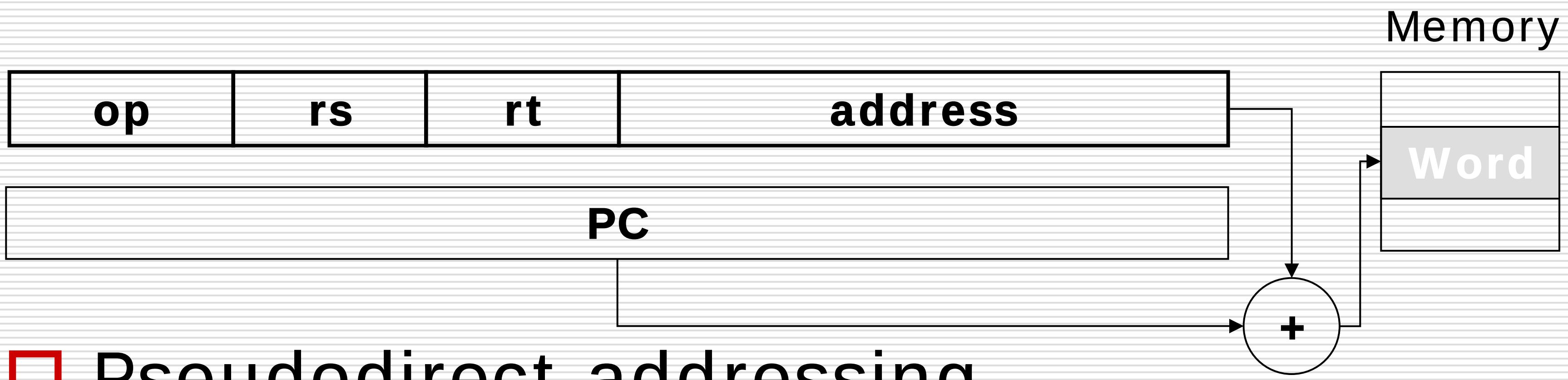
Addressing Modes

- Base addressing

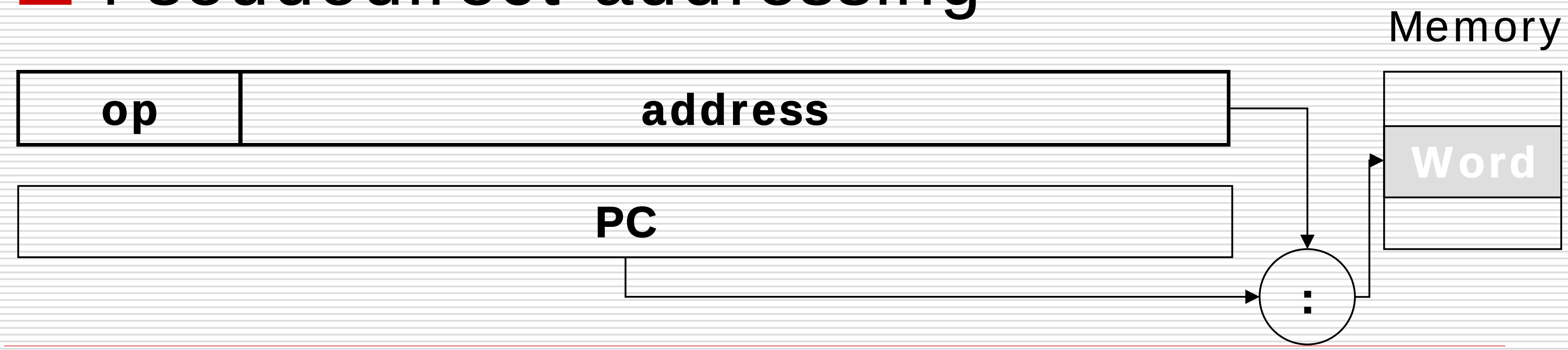


Addressing Modes

- PC-relative addressing



- Pseudodirect addressing



Decoding Machine Code

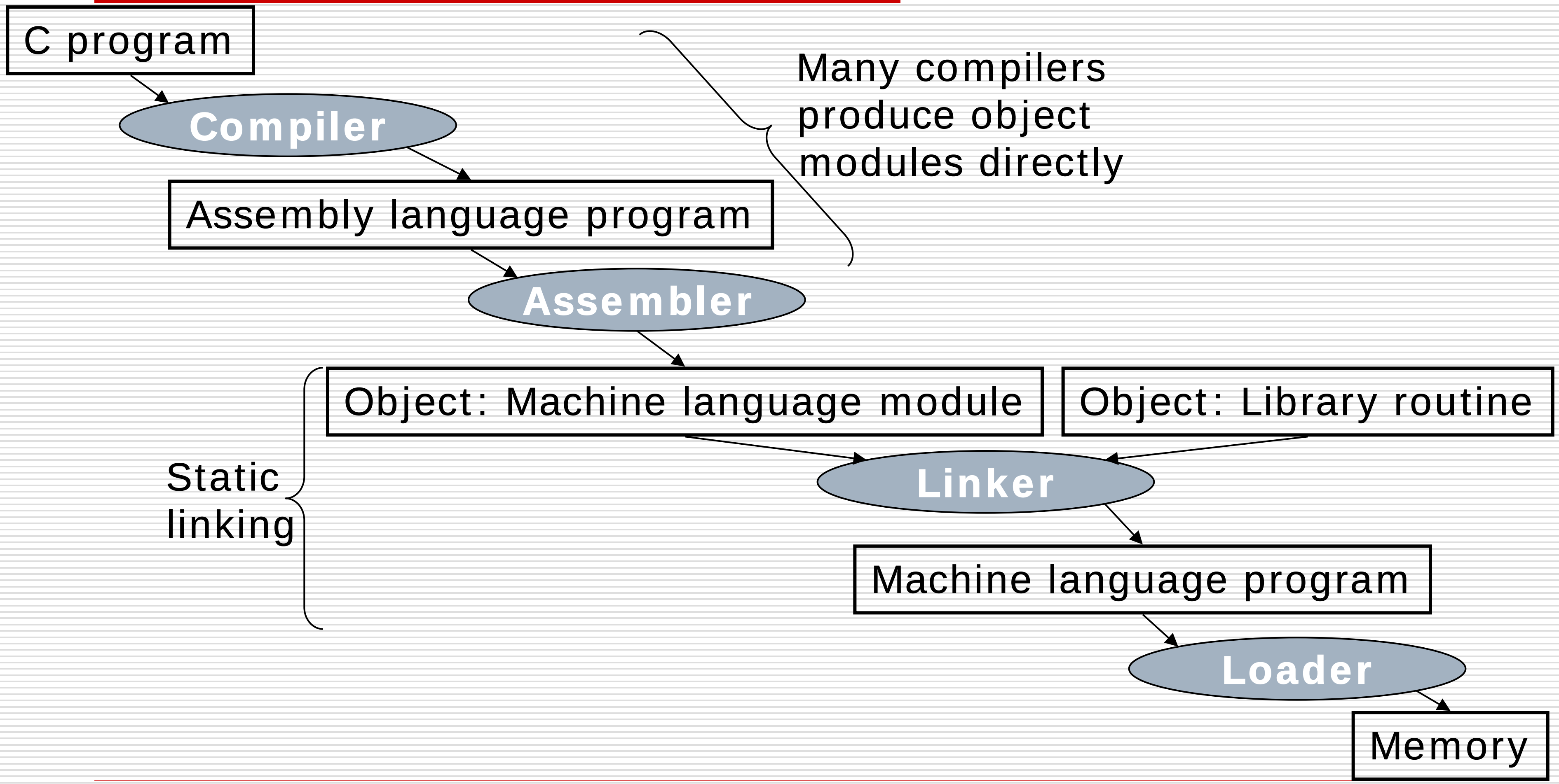
- What is the assembly language statement corresponding to this machine instruction?
  - 00af8020$_{hex}$
  - ➔ 0000 0000 1010 1111 1000 0000 0010 0000

  - op = 000000 ⇨ R-format
  - rs = 00101 (a1)/ rt = 01111 (t7)/ rd = 10000 (s0)
  - shamt = 00000 / funct = 100000 ⇨ add

  - ➔ add \$s0, \$a1, \$t7

MIPS instruction encoding@Fig.2.19@P.135
MIPS register conventions@Fig.2.14@P.121

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

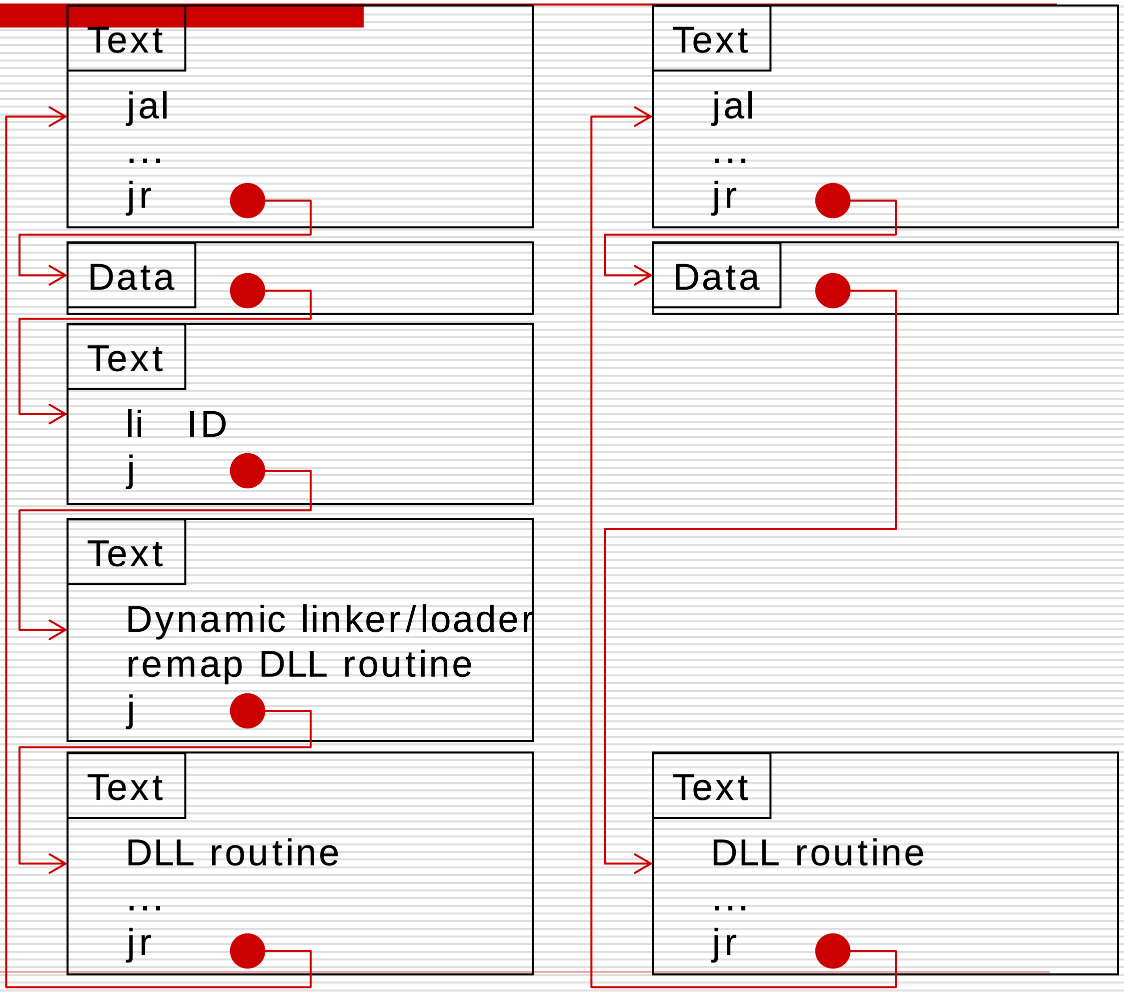
Lazy Linkage

Indirection table

Stub: Loads routine ID,
Jump to linker/loader

Linker/loader code

Dynamically
mapped code



First call
to DLL routine

Subsequent calls
to DLL 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

Array vs. Pointers in C

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


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

Array Version of Clear in MIPS

```
        add     $t0, $zero, $zero
loop1:  sll     $t1, $t0, 2
        add     $t2, $a0, $t1
        sw      $zero, 0($t2)
        addi    $t0, $t0, 1
        slt     $t3, $t0, $a1
        bne     $t3, $zero, loop1
```

Pointer Version of Clear in MIPS

```
        add     $t0, $a0, $zero
loop2:  sw      $zero, 0($t0)
        addi    $t0, $t0, 4
        sll     $t1, $a1, 2
        add     $t2, $a0, $t1
        slt     $t3, $t0, $t2
        bne     $t3, $zero, loop2
```

New Pointer Version of Clear

```
        add     $t0, $a0, $zero
        sll     $t1, $a1, 2
        add     $t2, $a0, $t1
loop2:  sw      $zero, 0($t0)
        addi    $t0, $t0, 4
        slt     $t3, $t0, $t2
        bne     $t3, $zero, loop2
```

Comparing the Two Versions

```
      add   $t0, $zero, $zero            add   $t0, $a0, $zero
lp1:  sll   $t1, $t0, 2                  sll   $t1, $a1, 2
      add   $t2, $a0, $t1                add   $t2, $a0, $t1
      sw    $zero, 0($t2)        lp2 :   sw    $zero, 0($t0)
      addi  $t0, $t0, 1                  addi  $t0, $t0, 4
      slt   $t3, $t0, $a1                slt   $t3, $t0, $t2
      bne   $t3, $zero, lp1              bne   $t3, $zero, lp2
```

Comparison of Array vs. Pointer

- 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

Summary: 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.
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.

Summary: MIPS Assembly Language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1, $s2, $s3	$s1=$s2+$s3	Three register operands
	subtract	sub $s1, $s2, $s3	$s1=$s2-$s3	Three register operands
	add immediate	addi $s1, $s2, 100	$s1=$s2+100	Used to add constants
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
Uncondi-tional 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

Summary: MIPS Assembly Language

Category	Instruction	Example	Meaning	Comments
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 half	lh \$s1, 100(\$s2)	\$s1=Memory[\$s2+100]	Halfword from memory to register
	store half	sh \$s1, 100(\$s2)	Memory[\$s2+100]=\$s1	Halfword 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 immed.	lui \$s1, 100	\$s1=100*2^{16}	Loads constant in upper 16 bits
Logical	and	and \$s1, \$s2, \$s3	\$s1=\$s2&\$s3	Bit-by-bit AND
	or	or \$s1, \$s2, \$s3	\$s1=\$s2|\$s3	Bit-by-bit OR
	nor	nor \$s1, \$s2, \$s3	\$s1=~(\$s2|\$s3)	Bit-by-bit NOR
	and immediate	andi \$s1, \$s2, 100	\$s1=\$s2&100	Bit-by-bit AND reg with constant
	or immediate	ori \$s1, \$s2, 100	\$s1=\$s2|100	Bit-by-bit OR reg with constant
	shift left logical	sll \$s1, \$s2, 10	\$s1=\$s2<<10	Shift left by constant
	shift right logical	srl \$s1, \$s2, 10	\$s1=\$s2>>10	Shift right by constant