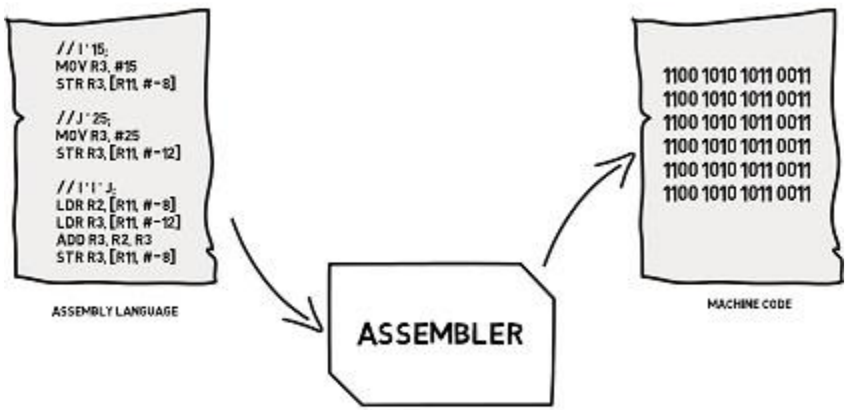
ساختار و زبان کامپیوتر

فصل چهارم

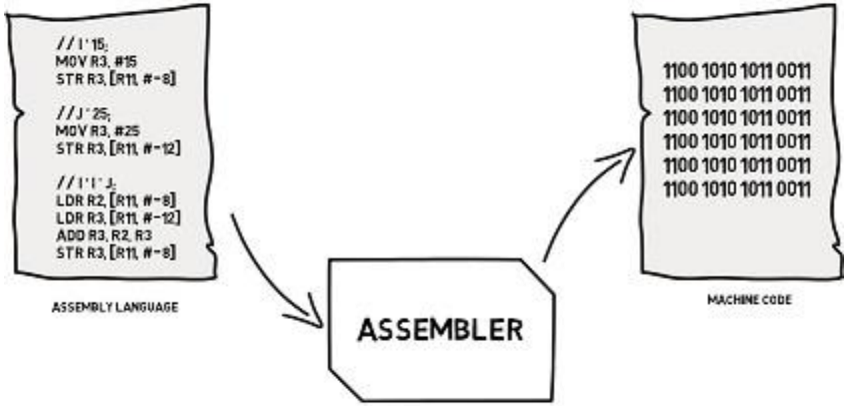
زبان اسمبلی MIPS-32



Computer Structure and Machine Language

Chapter Four

MIPS-32 Assembly Language



Copyright Notice

Parts (text & figures) of this lecture are adopted from:

- D. Patterson & J. Hennessey, "Computer Organization & Design, The Hardware/Software Interface", 6^{th} Ed., MK publishing, 2020

Contents

MIPS-32 Processor

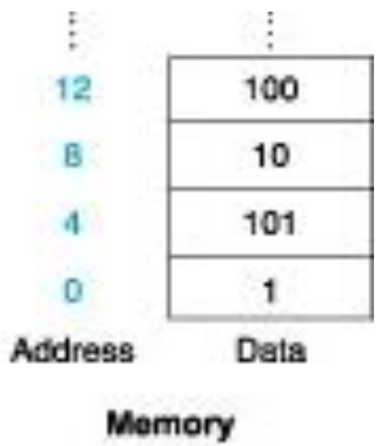
- 32-Bit Processor
  - Registers 32 bits
  - Arithmetic & logical operations 32 bits
- Load/Store ISA
  - Only load/store instructions can access memory
    - Arithmetic/logical instructions no access to memory
- 32-Bit Instruction Length

Memory Organization

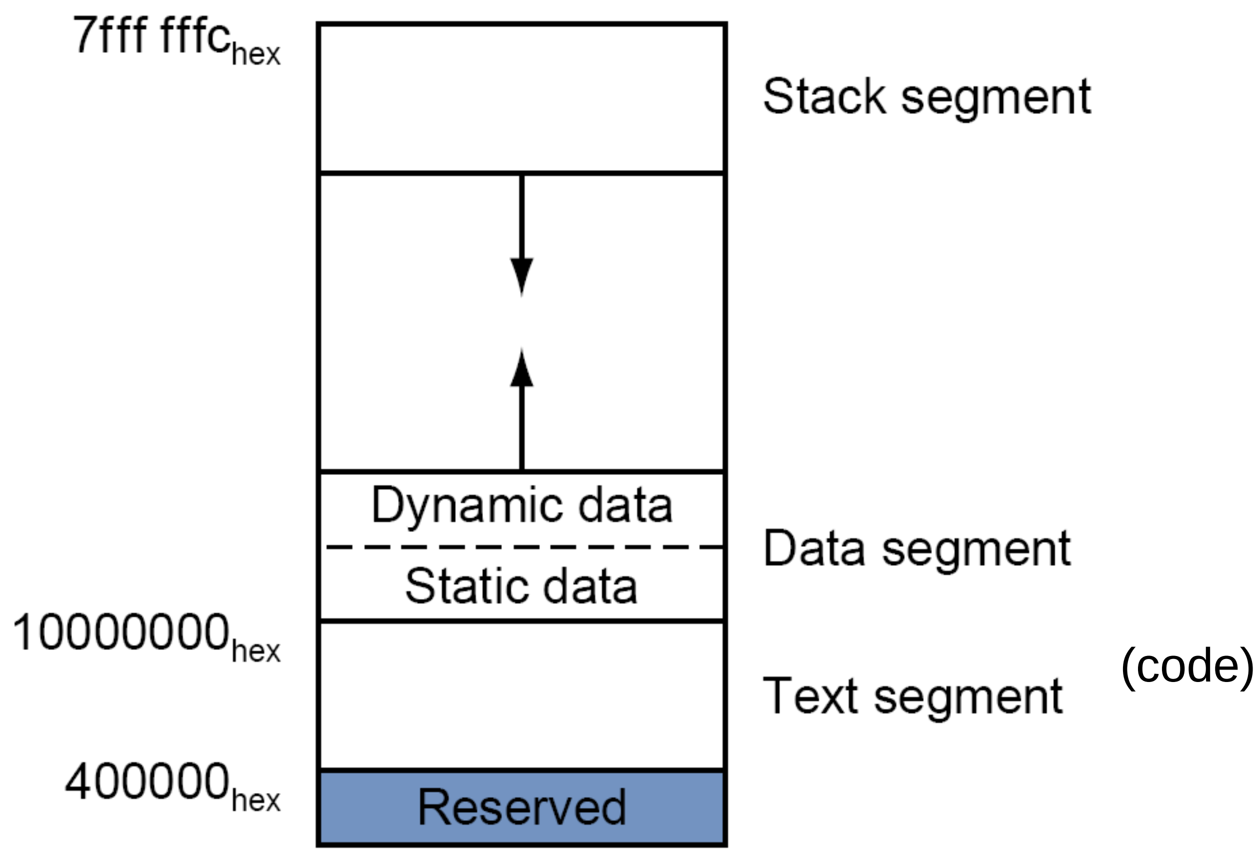
- Organized as array of bytes or words
  - One Byte = 8 bits
  - Byte is smallest addressable entry in memory
- Possible Organizations
  - Byte addressable, byte accessible ✓ for MIPS
  - Byte addressable, word accessible
  - Word addressable, word accessible

MIPS Memory Organization

- MIPS Uses Words (4 bytes)
  - Words start at addresses that are multiples of 4
  - This is called alignment restriction
  - Addresses are byte-based addressing



MIPS Memory Allocation



MIPS Registers

- $s0-$s7
  - General registers
  - Must be saved when calling subroutine
- $t0-$t9
  - Temporary registers
  - Local to each subroutine
- $a0-$a3
  - Arguments for subroutine call
- $v0-$v1
  - Values for results of a subroutine call

MIPS Registers (cont.)

Register Number	Mnemonic Name	Conventional Use		Register Number	Mnemonic Name	Conventional Use
$0	$zero	Permanently 0		$24, $25	$t8, $t9	Temporary
$1	$at	Assembler Temporary (reserved)		$26, $27	$k0, $k1	Kernel (reserved for OS)
$2, $3	$v0, $v1	Value returned by a subroutine		$28	$gp	Global Pointer
$4–$7	$a0–$a3	Arguments to a subroutine		$29	$sp	Stack Pointer
$8–$15	$t0–$t7	Temporary (not preserved across a function call)		$30	$fp	Frame Pointer
$16–$23	$s0–$s7	Saved registers (preserved across a function call)		$31	$ra	Return Address

Contents

MIPS Design Principles

- Simplicity favors regularity
- Smaller is faster
- Make the common case fast
- Good design demands good compromises

Simplicity Favors Regularity

- Regularity makes implementation simpler
- Simplicity enables higher performance at lower cost
- All arithmetic/logical instructions have three operands
  - two sources and one destination

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

Smaller is Faster

- Arithmetic instructions use register operands
  - cf. main memory with millions of locations
- 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

Make the Common Case Fast

- Constant data specified in an instruction

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

- No subtract immediate instruction
- Just use a negative constant

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

- Small constants are common
- Immediate operand avoids a load instruction

Good Design Demands Good Compromises

- Different formats complicate decoding, but allow 32-bit instructions uniformly
- Keep formats as similar as possible

6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
op	rs	rt	rd	shamt	funct
op	rs	rt	16 bit address		
op	26 bit address				

Contents

MIPS Instructions

- Arithmetic
- Logical & Shift
- Data Transfer
- Control
  - Conditional branch
  - Unconditional branch

Arithmetic Instructions

- Addition / Subtraction
  - `add r1,r2,r3`
    - *e.g.* `add $t0,$s2,$s4` *# $t0=$s2+$s4*
  - `sub r1,r2,r3`
    - *e.g.* `sub $t0,$s2,$s4` *# $t0=$s2-$s4*
  - `addi r1,r2,cnst`
    - *e.g.* `addi $t0,$s2,2` *# $t0=$s2+2*
    - *e.g.* `addi $t0,$s2,-2` *# $t0=$s2-2*

Logical and Shift Instructions

- Operate on bits individually by
  - selective set
  - selective reset
  - selective complement
- Use to isolate fields by
  - shift back and forth
  - selective mask

Logical Instructions

- And / Or / Negate
  - and r1,r2,r3
    - e.g. `and $t0,$s2,$s4    # $t0 = $s2 & $s4`
  - or r1,r2,r3
    - e.g. `or $t0,$s2,$s4    # $t0 = $s2 | $s4`
  - nor r1,r2,r3
    - e.g. `nor $t0,$s2,$s4    # $t0 = ~($s2|$s4)`
  - *What about negation?*
    - e.g. `nor $t0,$s2,$zero   # $t0=~$s2`

Shift Instructions

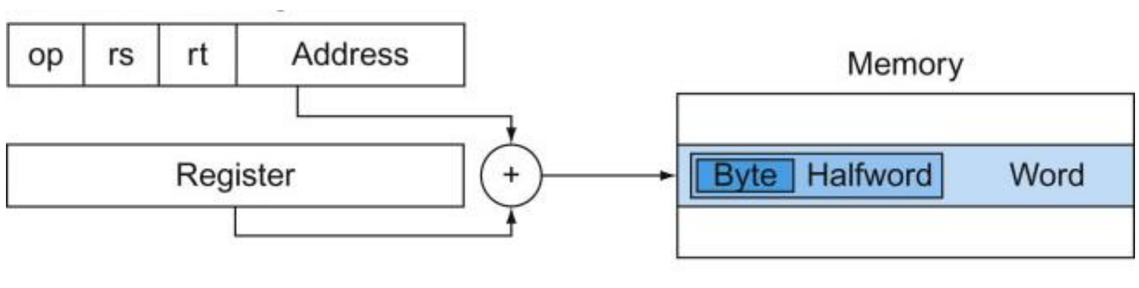
- Logical Shift
  - `sll r1,r2,nbits`
    - e.g. `sll $t0,$s2,3`   # $t0 = $s2 << 3
  - `srl r1,r2,nbits`
    - e.g. `srl $t0,$s2,2`   # $t0 = $s2 >> 2
  - …
  - Will see more shift instructions later

Data Transfer Instructions

- Between registers & memory
  - Transfer data from/to memory
  - Transfer memory address to register
  - Swap
- Stack operations
  - Push
  - Pop

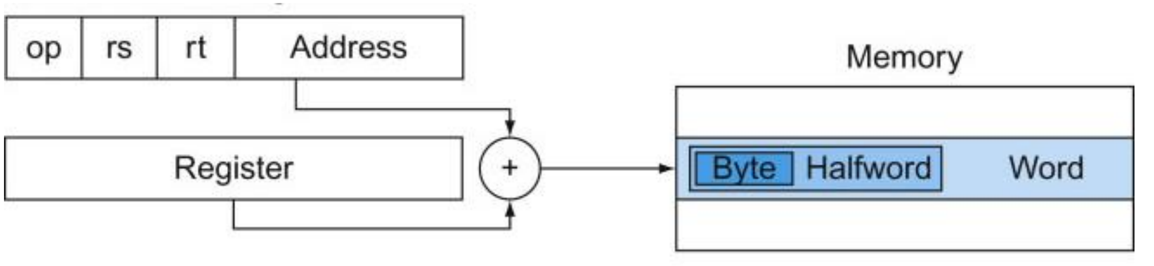
Transfer Data from/to Memory

- Load / Store
  - `lw r1,address(r2)` # load word from memory
    - `lw $s1,100($s2)` # $s1=Memory[$s2+100]
  - `sw r1,address(r2)` # store word to memory
    - `sw $s1,100($s2)` # Memory[$s2+100]=$s1



Base Addressing Mode

- Memory address in load and store specified by
  - a base register and an offset



MIPS Instructions: Example 1

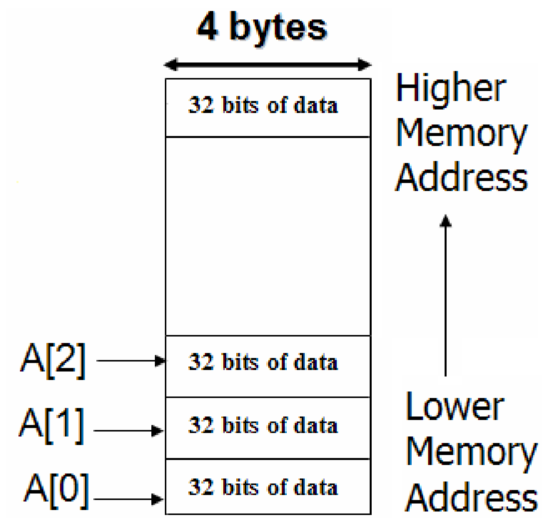
- compute $f = (g+h) - (i+j)$
  - Assumptions
    - g in $t0, h in $t1, i in $t2, j in $t3
    - f in $s0
  - Answer ?

MIPS Instructions: Example 1 (cont.)

- compute f = (g+h) - (i+j)
  - Assumptions
    - g in $t0, h in $t1, i in $t2, j in $t3
    - f in $s0
  - Answer
    - `add $s0,$t0,$t1`
    - `add $s1,$t2,$t3`
    - `sub $s0,$s0,$s1`

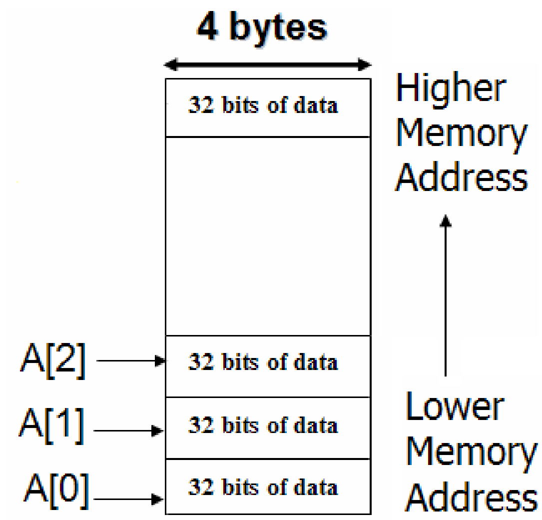
MIPS Instructions: Example 2

- A[12] = h + A[8]
  - Assumptions
    - Address of A in $s3
    - Variable h is in $s2
  - Answer ?



MIPS Instructions: Example 2 (cont.)

- A[12] = h + A[8]
  - Assumptions
    - Address of A in $s3
    - Variable h is in $s2
  - Answer
    - `lw    $t0,32($s3)    # A[8]`
    - `add   $t0,$s2,$t0    # h+A[8]`
    - `sw    $t0,48($s3)`



MIPS Instructions: Example 3

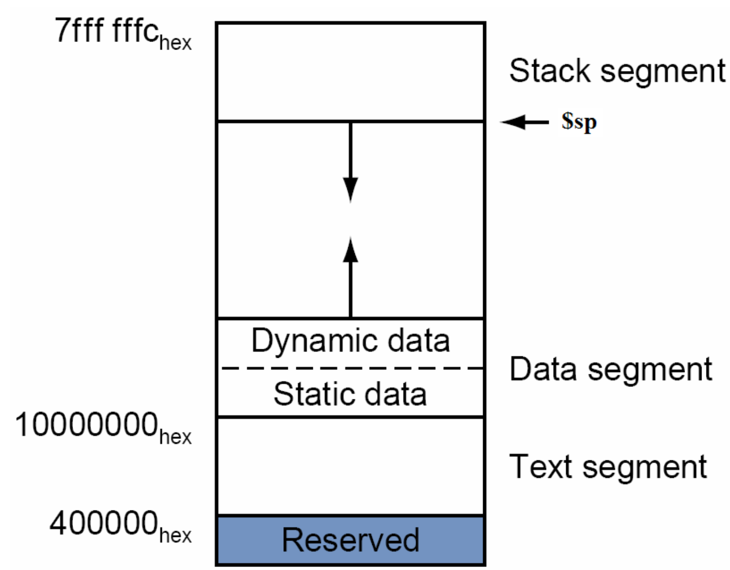
- compute A[4] as
  - A[4] = (A[0]+A[1]) - (A[2]+A[3])
- Assumptions
  - A is an array in main memory
  - Four-byte entry
  - Starting address of array A in $s0

MIPS Instructions: Example 3 (cont.)

- A[4] = (A[0]+A[1]) - (A[2]+A[3])
- Answer
  - `lw $t0,0($s0)`
  - `lw $t1,4($s0)`
  - `lw $t2,8($s0)`
  - `lw $t3,12($s0)`
  - `add $s1,$t0,$t1`
  - `add $s2, $t2,$t3`
  - `sub $s3,$s1,$s2`
  - `sw $s3,16($s0)`

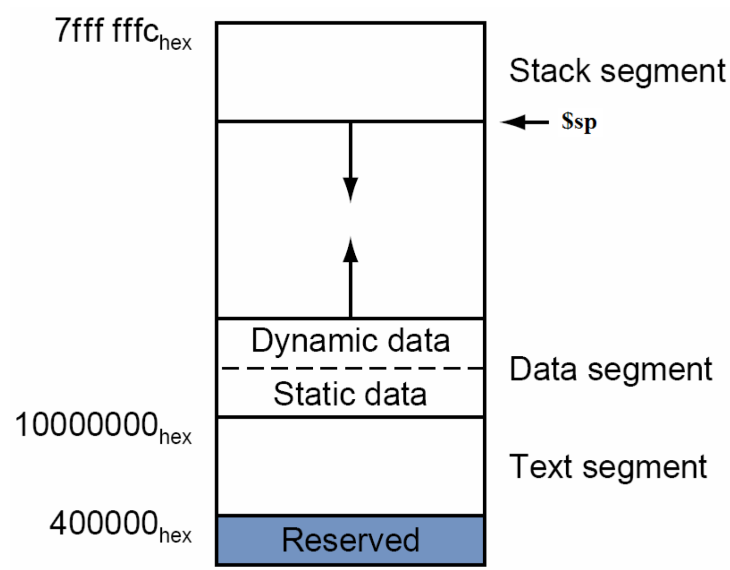
Stack Example

- Stack grows from higher to lower addresses
- $sp contains address of word on top of stack
- POP $s0
  - ?
- PUSH $s1
  - ?



Stack Example (cont.)

- Stack grows from higher to lower addresses
- $sp contains address of word on top of stack
- POP $s0 :
  - `lw $s0,0($sp)`
  - `addi $sp,$sp,4`
- PUSH $s1 :
  - `addi $sp,$sp,-4`
  - `sw $s1, 0($sp)`



Contents

MIPS Assembly Code

- Consists of MIPS instructions and data
- Instructions given in .text segments
  - A program may have multiple .text segments
- Data defined in .data segments by directives
  - .word defines 32 bit numbers
  - .space defines n number of bytes
  - .asciiz defines a string
  - …

First MIPS Assembly Program

- Compute sum of five homework assignment scores
- Scores are in memory
  - `scores: .word 95, 87, 98, 100, 100`
  - `sum:    .space 4`
- We need to:
  - Load address of scores to a register
  - Load first two scores
  - Add them
  - Load third score and add it to the sum and so on

Add Five Scores – 1st version

```
.text
        la $s0,scores    # $s0 has the address of the scores
        lw $t0,0($s0)    # the first score
        lw $t1,4($s0)    # the 2nd score
        add $t0,$t0,$t1
        lw $t1,8($s0)    # the 3rd score
        add $t0, $t0,$t1
        lw $t1,12($s0)   # the 4th score
        add $t0,$t0,$t1
        lw $t1,16($s0)   # the 5th score
        add $t0,$t0,$t1
        la $s1,sum
        sw $t0,0($s1)
.data
scores:
        .word 95, 87, 98, 100, 100
sum:
        .space 4
```

Contents

System Calls

- MIPS Assembly Notation
  - syscall
- Provided by MIPS Assembly simulators
  - Small set of OS-like services
- How to Call?
  - Initialize $v0 and $a0
  - Then, “syscall”

System Call (cont.)

Service	System call code	Arguments	Result
print_int	1	$a0 = integer	
print_float	2	$f12 = float	
print_double	3	$f12 = double	
print_string	4	$a0 = string	
read_int	5		integer (in $v0)
read_float	6		float (in $f0)
read_double	7		double (in $f0)
read_string	8	$a0 = buffer, $a1 = length	
sbrk	9	$a0 = amount	address (in $v0)
exit	10		
print_char	11	$a0 = char	
read_char	12		char (in $v0)
open	13	$a0 = filename (string), $a1 = flags, $a2 = mode	file descriptor (in $v0)
read	14	$a0 = file descriptor, $a1 = buffer, $a2 = length	num chars read (in $a0)
write	15	$a0 = file descriptor, $a1 = buffer, $a2 = length	num chars written (in $a0)
close	16	$a0 = file descriptor	
exit2	17	$a0 = result	

Add Two Scores

```
.text
        la $s0,scores    # $s0 has the address of the scores
        lw $t0,0($s0)    # the first score
        lw $t1,4($s0)    # the 2nd score
        add $t0,$t0,$t1
        la $s1,sum
        sw $t0,0($s1)
        li $v0,4
        la $a0,msg
        syscall          # printout the message
        li $v0,1
        la $s0,sum
        lw $a0,0($s0)
        syscall          # printout the sum
        li $v0,4
        la $a0,nw
        syscall          # printout newline
        li $v0,10
        syscall          # exit
.data
scores:
        .word 12, 19
sum:    .space 4
msg:    .asciiz "Sum of two scores is: "
nw:     .asciiz "\n\r"
```

Contents

Decisions Making Instructions

- A distinctive feature of programs is that they can make decisions based on input data
- To support decision making, MIPS has two conditional branch instructions
  - similar to an "if" statement with a goto
- MIPS has also an unconditional branch,
  - equivalent to goto in C

Jump

- `j L1`
  - *Unconditional jump*
  - *Jump to instruction labeled with L1*
  - *In C, it is equivalent to*
    - `goto L1`

Branch if Equal

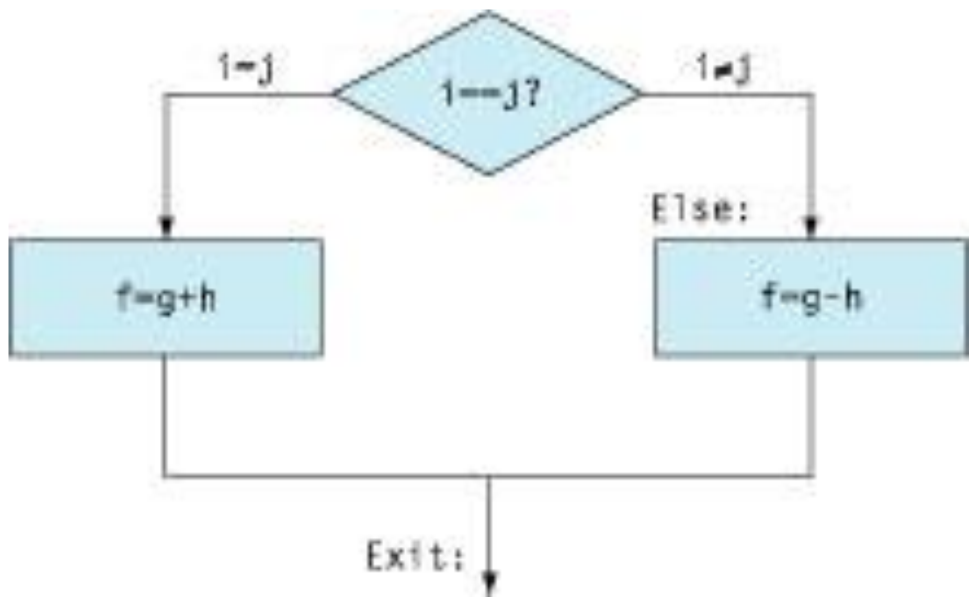
- `beq r1,r2,L1`
  - Conditional branch
  - comparing values in r1 and r2
  - go to L1 if values are equal
    - L1 is a label
  - In C, it is equivalent to
    - `if (r1 == r2) goto L1`

Branch if not Equal

- `bne r1,r2,L1`
  - Conditional branch
  - comparing values in r1 and r2
  - go to L1 if values are not equal
    - L1 is a label
  - In C, it is equivalent to
    - `if (r1 != r2) goto L1`

"if then else" Example

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



"if then else" Example (cont.)

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

Variables f, g, h, i, j are in registers $s0 through $s4

MIPS Code ?

“if then else” Example (cont.)

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

Variables f, g, h, i, j are in registers $s0 through $s4

```
    if (i != j)
       goto Else;
    f = g + h;
    goto Exit;
Else:
    f = g - h;
Exit:
```

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

Loop Example

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

i, k & address of save are in registers $s3, $s5, $s6

MIPS
Code?

Loop Example

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

i, k & address of save are in registers $s3, $s5, $s6

```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: …
```

Comparison Instructions

- `slt reg1,reg2,reg3` *(set on less than)*
  - *e.g.* `slt $t0,$s3,$s4`
    - *if ($s3 < $s4) ➔ $t0 =1*
    - *otherwise $t0 =0*
- `slti reg1,reg2,cnst` *(slt immediate)*
  - *e.g.* `slti $t0,$s3,10`
    - *if ($s3 < 10) ➔ $t0 =1*
    - *otherwise $t0 =0*

Other Conditional Branches

- So far bne & beq
- How to implement all relative conditions?
  - less than
  - less than or equal
  - greater than
  - greater than or equal
- Use `slt` in combination with bne & beq

Other Conditional Branches (cont.)

- Branch on "less than"

?

Other Conditional Branches (cont.)

- Branch on "less than"

```
slt $t0,$s1,$s2 #if ($s1<$s2) $t0=1
bne $t0,$zero,L #if ($t0!=0) goto L
```

- Branch on "greater than or equal"

?

Other Conditional Branches (cont.)

- Branch on "less than"

```
slt $t0,$s1,$s2   #if ($s1<$s2) $t0=1
bne $t0,$zero,L   #if ($t0!=0) goto L
```

- Branch on "greater than or equal"

```
slt $t0,$s1,$s2   #if ($s1<$s2) $t0=1
beq $t0,$zero,L   #if ($t0=0) goto L
```

Discussion

- Why MIPS designers didn't include all possible relative conditions in MIPS ISA?

Other Comparison Instructions

- 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`
    - $+4{,}294{,}967{,}295 > +1 \Rightarrow \$t0 = 0$

Contents

Procedures and Functions

- Question?
  - Why use procedure/function/subroutine?
- Answer:
  - Programmers use procedures to
    - structure and organize programs
    - make them easier to understand
    - allow code to be reused

Call/ Return Instructions

- `jal label` *(jump and link)*
  - *e.g.* `jal sub1`
    - Jump to sub1
    - Save return address in $ra
- `jr reg` *(jump to register)*
  - *e.g.* `jr $ra`
    - Jump to the address saved in $ra

MIPS Calling Conventions

- $a0 - $a3:
  - Four argument registers in which to pass parameters
- $v0 - $v1
  - Two value registers in which to return values
- $ra:
  - One return address register to return to point of origin

Procedure Execution Flow

- Program must follow these Steps:
  - Place parameters in a place where procedure can access them
  - Transfer control to procedure
  - Acquire storage resources needed for procedure
  - ✓ Perform desired tasks
  - Place results in a place where calling program can access them
  - Return control to point of origin

A Simple Example (in C)

```c
#include <stdio.h>
int leaf_example(int g, int h, int i, int j)
{
  int f;
  f = (g + h) - (i + j);
  return f;
}

int main(int agrc, char *argv[])
{
  int f, g, h, i, j;

  g = 5; h = -20; i = 13; j = 3;
  f = leaf_example(g, h, i, j);
  printf("\nThe value of f is %d.\n", f);
  return 0;
}
```

A Simple Example: main

```
.text
        la $t0, g
        lw $a0, 0($t0)              # $a0=g
        lw $a1, 4($t0)              # $a1=h
        lw $a2, 8($t0)              # $a2=i
        lw $a3, 12($t0)             # $a3=j
        jal leaf_example            # call procedure
        la $t0, f
        sw $v0, 0($t0)              # f=$v0
.data
g:      .word 5,-20,13,3            #g,h,i,j
f:      .space 4
```

A Simple Example: leaf

```
.globl leaf_example
leaf_example:

        add  $t0, $a0, $a1        #register $t0 contains g + h
        add  $t1, $a2, $a3        #register $t1 contains i + j
        sub $s0, $t0, $t1         #f = (g + h) - (i + j)
        add $v0, $s0, $0          #returns f

        jr $ra                    #return to calling program
```

What else?

```
.globl leaf_example
leaf_example:
```

What do we need to do in order to guarantee correctness of program?

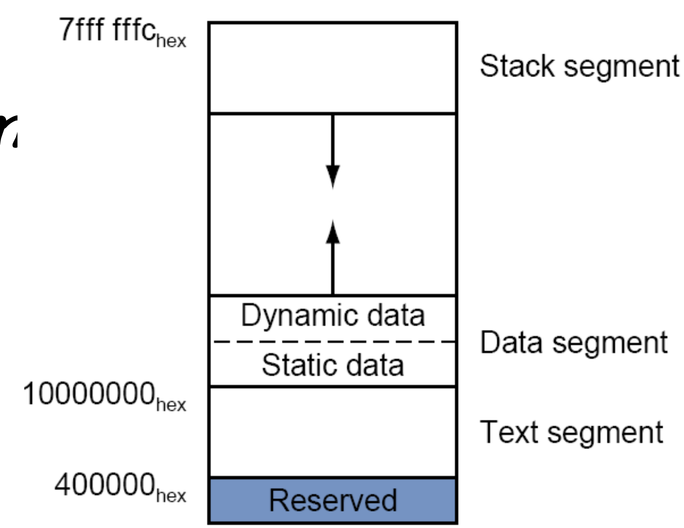
```
add  $t0, $a0, $a1        #register $t0 contains g + h
add  $t1, $a2, $a3        #register $t1 contains i + j
sub $s0, $t0, $t1         #f = (g + h) - (i + j)
add $v0, $s0, $0          #returns f



jr $ra                    #return to calling program
```

Register Spilling

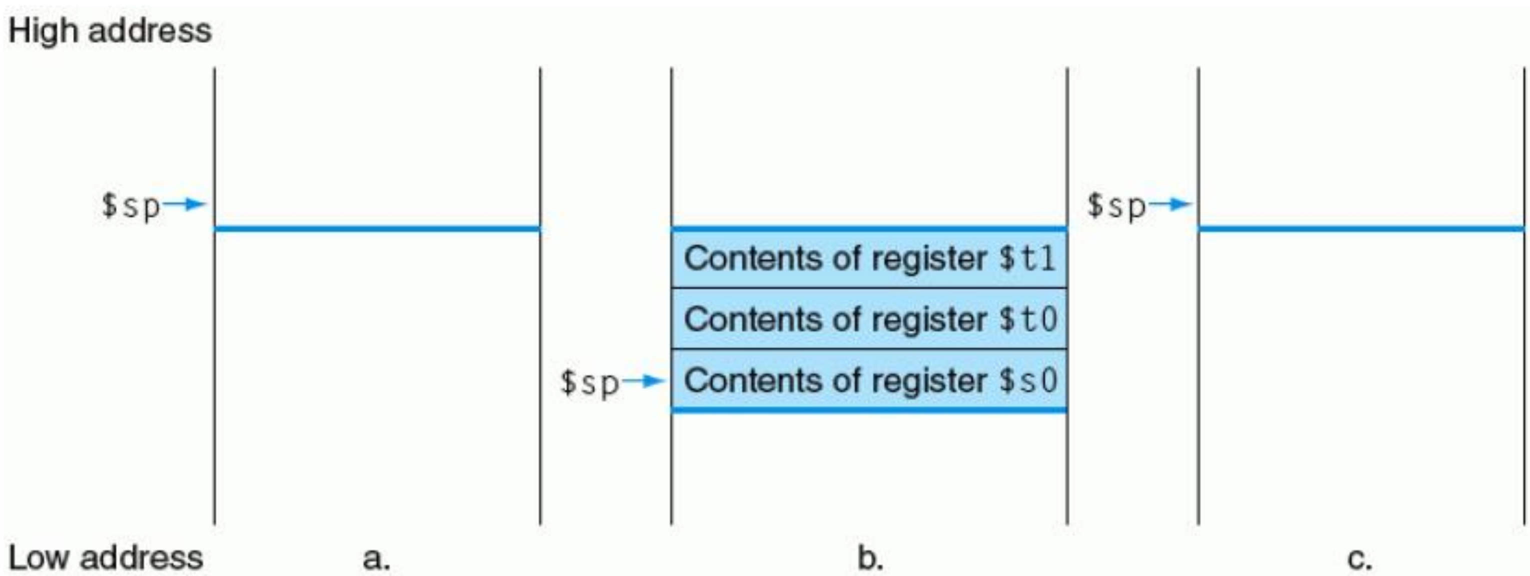
- Callee has to save all registers it uses and restore values before it returns
  - By storing them on stack
    - At the beginning
  - Then restoring them
    - At the end



A Simple Example (complete version)

```
.globl leaf_example
leaf_example:
        addi $sp, $sp, -12        #make space on stack
        sw   $t1, 8($sp)          #save $t1
        sw   $t0, 4($sp)          #save $t2
        sw   $s0, 0($sp)          #save $s0
        add  $t0, $a0, $a1        #register $t0 contains g + h
        add  $t1, $a2, $a3        #register $t1 contains i + j
        sub $s0, $t0, $t1         #f = (g + h) - (i + j)
        add $v0, $s0, $0          #returns f
        lw      $s0, 0($sp)       #restore $s0
        lw      $t0, 4($sp)       #restore $t0
        lw      $t1, 8($sp)       #restore $t1
        addi $sp, $sp, 12         #adjust stack pointer
        jr $ra                    #return to calling program
```

Stack Pointer



Nested Procedures

- Procedures that do not call others are called *leaf* procedures
- Procedures may invoke other procedures
- Caller:
  - The procedure that calls another procedure
- Callee:
  - The procedure that is called by another procedure

MIPS General Registers

- MIPS divides 18 registers into two groups
  - $t0 - $t9
    - 10 temporary registers not preserved by callee on a procedure call
    - Caller-saved registers
      - Caller must save those it is using
  - $s0 - $s7
    - 8 saved registers must be preserved on a procedure call
    - Callee-saved registers
      - Callee must save those it is going to use

Caller Must Do (before…)

- Before it calls a subroutine, it must:
  - Save caller-saved registers on stack
    - It includes \$a0 - \$a3, \$t0 - \$t9, and \$ra
    - Why \$a0 - \$a3, \$ra?
  - Pass parameters
    - Up to four parameters passed by \$a0 - \$a3
  - Execute a jal instruction
    - Jumps to callee's first instruction and save address of next instruction in \$ra

Caller Must Do (after…)

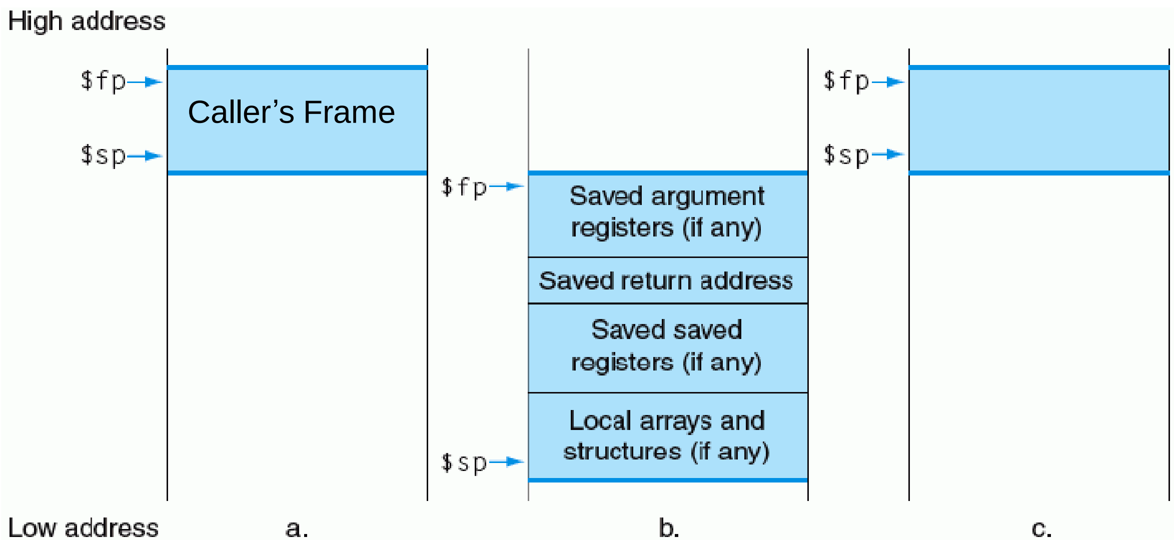
- After subroutine call, it needs to
  - Read returned values from $v0 and $v1
  - Restore caller-saved registers

Callee Must Do (at first)

- Callee must do the following:
  - Allocate memory for its frame by subtracting its frame size from stack pointer
  - Save callee-saved registers in frame
    - It must save \$s0 - \$s7, \$fp before changing them
    - \$ra needs to be saved if callee itself makes a call
  - When needed, update frame pointer
    - In this case, \$fp must be saved

Procedure Frame (Activation Record)

Segment of stack containing a procedure's saved registers and local variables



We can avoid using $fp by avoiding changes to $sp within a procedure: adjust the stack only on entry and exit of the procedure

Callee Must Do (at the end)

- Before it returns to caller:
  - Place return values in $v0 & $v1 (if needed)
  - Restore all callee-saved registers
    - That were saved at procedure entrance
  - Pop stack frame
    - By adding frame size to $sp
  - Return by jumping to address in $ra
    - Using jr $ra

Earlier Example

```
.globl leaf_example
leaf_example:
      addi $sp, $sp, -12        #make space on stack
      sw   $t1, 8($sp)          #save $t1
      sw   $t0, 4($sp)          #save $t2
      sw   $s0, 0($sp)          #save $s0
      add  $t0, $a0, $a1        #register $t0 contains g + h
      add  $t1, $a2, $a3        #register $t1 contains i + j
      sub $s0, $t0, $t1         #f = (g + h) – (i + j)
      add $v0, $s0, $0          #returns f
      lw      $s0, 0($sp)       #restore $s0
      lw      $t0, 4($sp)       #restore $t0
      lw      $t1, 8($sp)       #restore $t1
      addi $sp, $sp, 12         #adjust stack pointer
      jr $ra                    #return to calling program
```

Revised Version

```
.globl leaf_example
leaf_example:
    addi $sp, $sp, -4          #make space on stack
    sw  $s0, 0($sp)            #save $s0
    add  $t0, $a0, $a1         #register $t0 contains g + h
    add  $t1, $a2, $a3         #register $t1 contains i + j
    sub $s0, $t0, $t1          #f = (g + h) - (i + j)
    add $v0, $s0, $0           #returns f
    lw   $s0, 0($sp)           #restore $s0
    addi $sp, $sp, 4           #adjust stack pointer
    jr $ra                     #return to calling program
```

Contents

MIPS Multiplication

- Two 32-bit registers for product
  - HI: most-significant 32 bits
  - LO: least-significant 32-bits
- Instructions
  - `mult rs,rt / multu rs,rt`
    - 64-bit product in HI/LO
  - `mfhi rd / mflo rd`
    - Move from HI/LO to rd
- Can test HI to see if product overflows 32 bits

MIPS Division

- Use HI/LO registers for result
  - HI: 32-bit remainder
  - LO: 32-bit quotient
- Instructions
  - `div rs,rt / divu rs,rt`
    - Result in HI/LO
  - `mfhi rd / mflo rd`
    - Move from HI/LO to rd
- No overflow or divide-by-0 checking
- Software must perform checks if required

Floating Point

- Representation for non-integral numbers
  - Including very small and very large numbers
- Like scientific notation
  - -2.34×10^{56} ← normalized
  - $+0.002 \times 10^{-4}$ ← not normalized
  - $+987.02 \times 10^{9}$ ← not normalized
- In binary
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
- Types `float` and `double` in C

Floating Point Standard

- Defined by IEEE Std 754-1985
- Developed in response to divergence of representations
  - Portability issues for scientific code
- Now almost universally adopted
- Two representations
  - Single precision (32-bit)
  - Double precision (64-bit)

IEEE Floating-Point Format

	single: 8 bits double: 11 bits	single: 23 bits double: 52 bits
S	Exponent	Fraction

$$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

- S: sign bit ($0 \Rightarrow$ non-negative, $1 \Rightarrow$ negative)
- Normalize significand: 1.0 ≤ |significand| < 2.0
  - Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden bit)
  - Significand is Fraction with the “1.” restored
- Exponent: excess representation: actual exponent + Bias
  - Ensures exponent is unsigned
  - Single: Bias = 127; Double: Bias = 1203

Single-Precision Range

- Exponents 00000000 and 11111111 reserved
- Smallest value
  - Exponent: 00000001
    $\Rightarrow$ actual exponent = 1 – 127 = –126
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- Largest value
  - exponent: 11111110
    $\Rightarrow$ actual exponent = 254 – 127 = +127
  - Fraction: 111…11 $\Rightarrow$ significand ≈ 2.0
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

Double-Precision Range

- Exponents 0000…00 and 1111…11 reserved
- Smallest value
  - Exponent: 00000000001
    $\Rightarrow$ actual exponent = 1 – 1023 = –1022
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- Largest value
  - Exponent: 11111111110
    $\Rightarrow$ actual exponent = 2046 – 1023 = +1023
  - Fraction: 111…11 $\Rightarrow$ significand ≈ 2.0
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Floating-Point Precision

- Relative precision
  - all fraction bits are significant
  - Single: approx 2^{-23}
    - Equivalent to $23 \times \log_{10}2 \approx 23 \times 0.3 \approx 6$ decimal digits of precision
  - Double: approx 2^{-52}
    - Equivalent to $52 \times \log_{10}2 \approx 52 \times 0.3 \approx 16$ decimal digits of precision

Floating-Point Example

- Represent –0.75
  - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - S = 1
  - Fraction = $1000\ldots00_2$
  - Exponent = –1 + Bias
    - Single: –1 + 127 = 126 = 01111110_2
    - Double: –1 + 1023 = 1022 = 01111111110_2
- Single: 1011111101000…00
- Double: 1011111111101000…00

Floating-Point Example

- What number is represented by the single-precision float

  11000000101000…00

  - S = 1
  - Fraction = $01000\ldots00_2$
  - Fxponent = $10000001_2 = 129$
- $x = (-1)^1 \times (1 + 01_2) \times 2^{(129-127)}$

  $= (-1) \times 1.25 \times 2^2$

  $= -5.0$

Denormal Numbers

- Exponent = 000...0 $\Rightarrow$ hidden bit is 0

$$x = (-1)^S \times (0 + \text{Fraction}) \times 2^{-\text{Bias}}$$

- Smaller than normal numbers
  - allow for gradual underflow, with diminishing precision
- Denormal with fraction = 000...0

$$x = (-1)^S \times (0 + 0) \times 2^{-\text{Bias}} = \pm 0.0$$

Two representations of 0.0!

Infinities and NaNs

- Exponent = 111...1, Fraction = 000...0
  - ±Infinity
  - Can be used in subsequent calculations, avoiding need for overflow check
- Exponent = 111...1, Fraction ≠ 000...0
  - Not-a-Number (NaN)
  - Indicates illegal or undefined result
    - e.g., 0.0 / 0.0
  - Can be used in subsequent calculations

IEEE 754 Standard: Summary

- Single precision (32bits)/Double precision (64bits)
- Normalized/ Denormalized forms
- Standard definitions for zero, infinity, NaN
- Check: https://www.h-schmidt.net/FloatConverter/IEEE754.html

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	± denormalized number
1–254	Anything	1–2046	Anything	± floating-point number
255	0	2047	0	± infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

MIPS Floating Point Support

- FP hardware is coprocessor 1
  - Adjunct processor that extends the ISA
- Separate FP registers
  - 32 single-precision: $f0, $f1, … $f31
  - Paired for double-precision: $f0/$f1, $f2/$f3, …
- FP instructions operate only on FP registers
  - Programs generally don't do integer ops on FP data, or vice versa
- FP load and store instructions
  - `lwc1`, `ldc1`, `swc1`, `sdc1`
    - e.g., `ldc1 $f8, 32($sp)`

MIPS Floating Point Instructions

Category	Instruction	Example	Meaning	Comments
Arithmetic	FP add single	`add.s $f2,$f4,$f6`	`$f2 = $f4 + $f6`	FP add (single precision)
	FP subtract single	`sub.s $f2,$f4,$f6`	`$f2 = $f4 - $f6`	FP sub (single precision)
	FP multiply single	`mul.s $f2,$f4,$f6`	`$f2 = $f4 × $f6`	FP multiply (single precision)
	FP divide single	`div.s $f2,$f4,$f6`	`$f2 = $f4 / $f6`	FP divide (single precision)
	FP add double	`add.d $f2,$f4,$f6`	`$f2 = $f4 + $f6`	FP add (double precision)
	FP subtract double	`sub.d $f2,$f4,$f6`	`$f2 = $f4 - $f6`	FP sub (double precision)
	FP multiply double	`mul.d $f2,$f4,$f6`	`$f2 = $f4 × $f6`	FP multiply (double precision)
	FP divide double	`div.d $f2,$f4,$f6`	`$f2 = $f4 / $f6`	FP divide (double precision)
Data transfer	load word copr. 1	`lwc1 $f1,100($s2)`	`$f1 = Memory[$s2 + 100]`	32-bit data to FP register
	store word copr. 1	`swc1 $f1,100($s2)`	Memory[`$s2` + 100] = `$f1`	32-bit data to memory
Condi-tional branch	branch on FP true	`bc1t 25`	if (cond == 1) go to PC + 4 + 100	PC-relative branch if FP cond.
	branch on FP false	`bc1f 25`	if (cond == 0) go to PC + 4 + 100	PC-relative branch if not cond.
	FP compare single (eq,ne,lt,le,gt,ge)	`c.lt.s $f2,$f4`	if (`$f2 < $f4`) cond = 1; else cond = 0	FP compare less than single precision
	FP compare double (eq,ne,lt,le,gt,ge)	`c.lt.d $f2,$f4`	if (`$f2 < $f4`) cond = 1; else cond = 0	FP compare less than double precision

Contents

MIPS Instruction Encoding

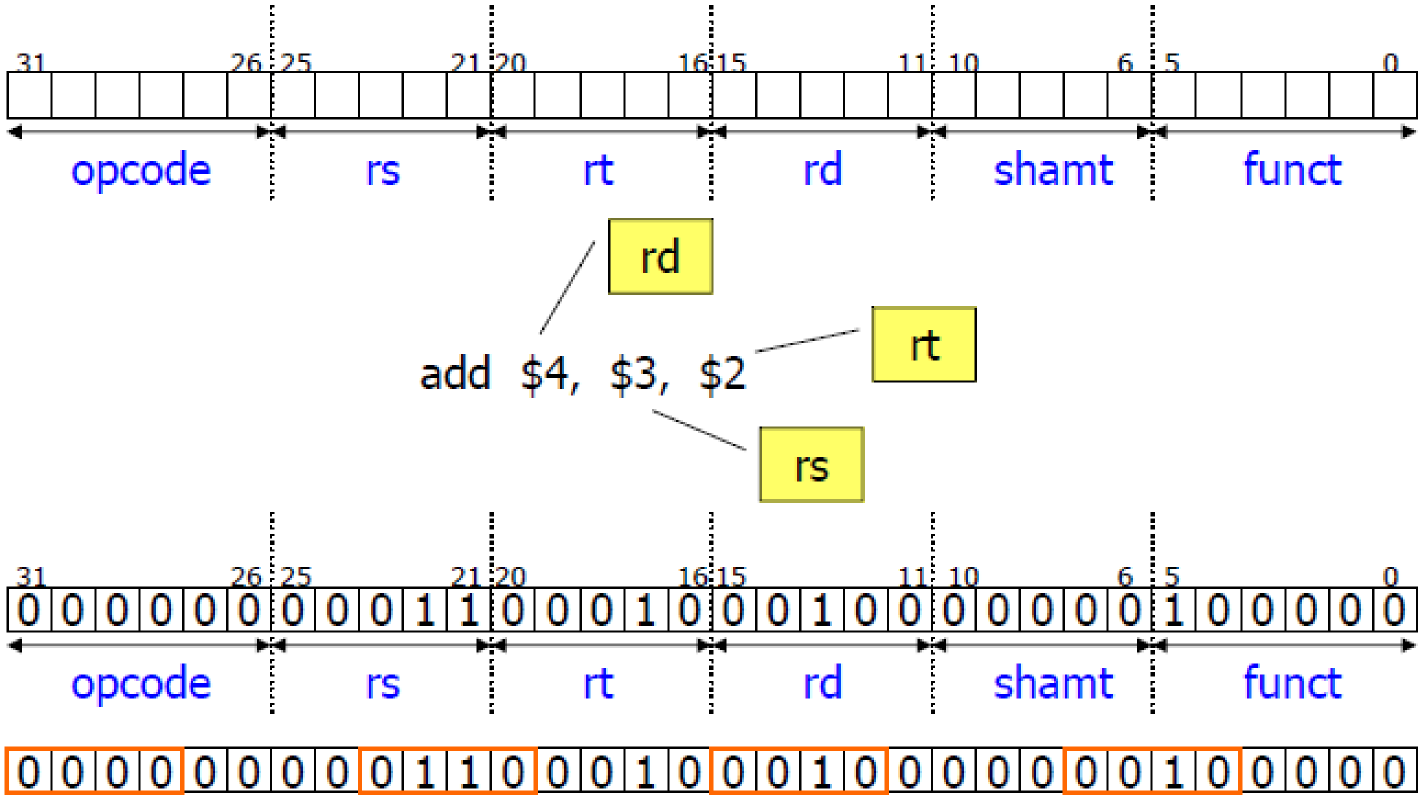
- MIPS instruction is exactly 32 bits
  - R-type (Register type)
  - I-type (Immediate type)
  - J-type (Jump type)

op	rs	rt	rd	shamt	funct
op	rs	rt	16 bit address		
op	26 bit address				

Instruction Encoding: Examples

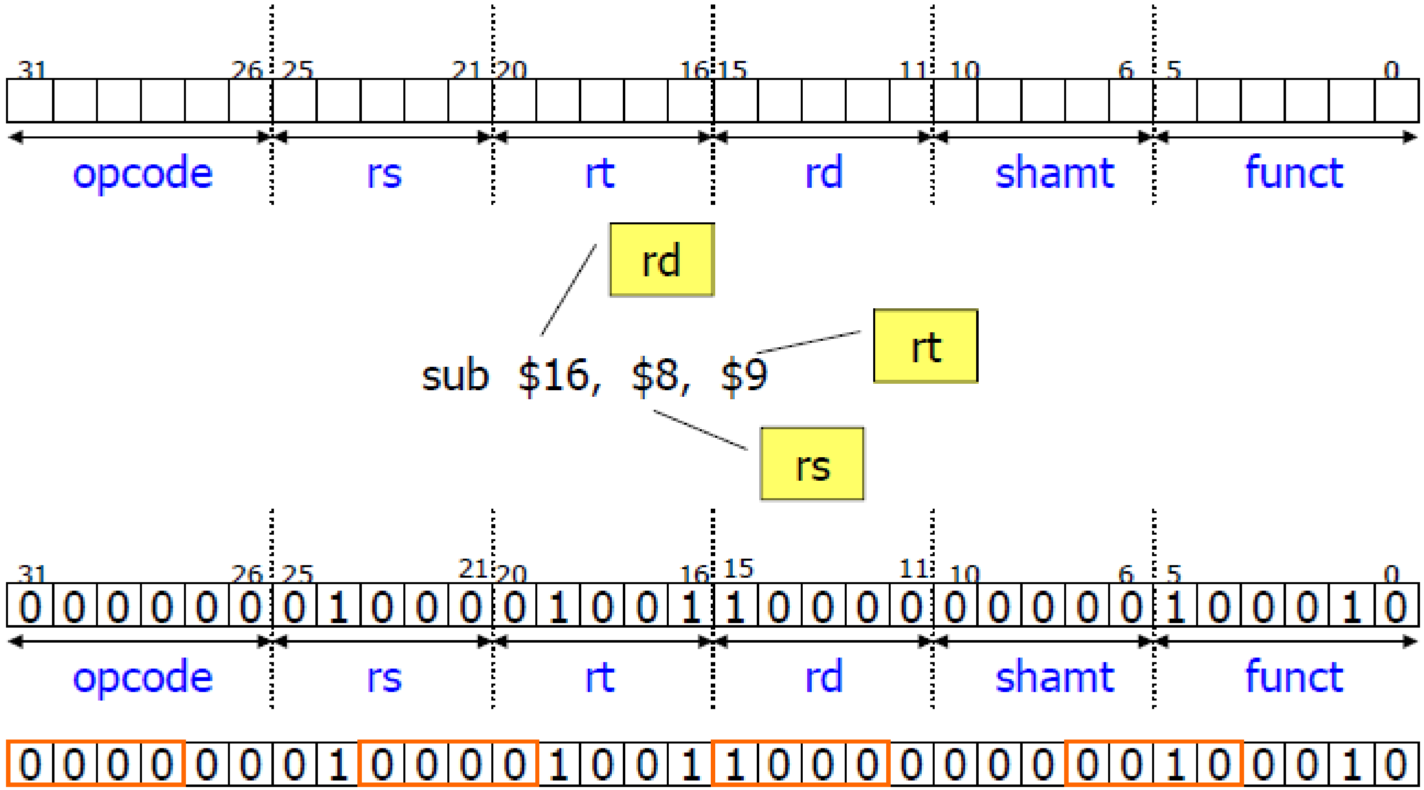
Instruction	Format	Op	rs	rt	rd	shamt	funct
add	R	0	reg	reg	reg	0	32_{ten}
sub (subtract)	R	0	reg	reg	reg	0	34_{ten}
sll (logical shift left)	R	0	0	reg	reg	shamt	0
Instruction	**Format**	**Op**	**rs**	**rt**	**constant/address**		
add immediate	I	8_{ten}	reg	reg	constant		
lw (load word)	I	35_{ten}	reg	reg	address		
sw (store word)	I	43_{ten}	reg	reg	address		
Instruction	**Format**	**Op**	**address**				
j (jump)	J	2_{ten}	address				

R-Type Encoding (add)



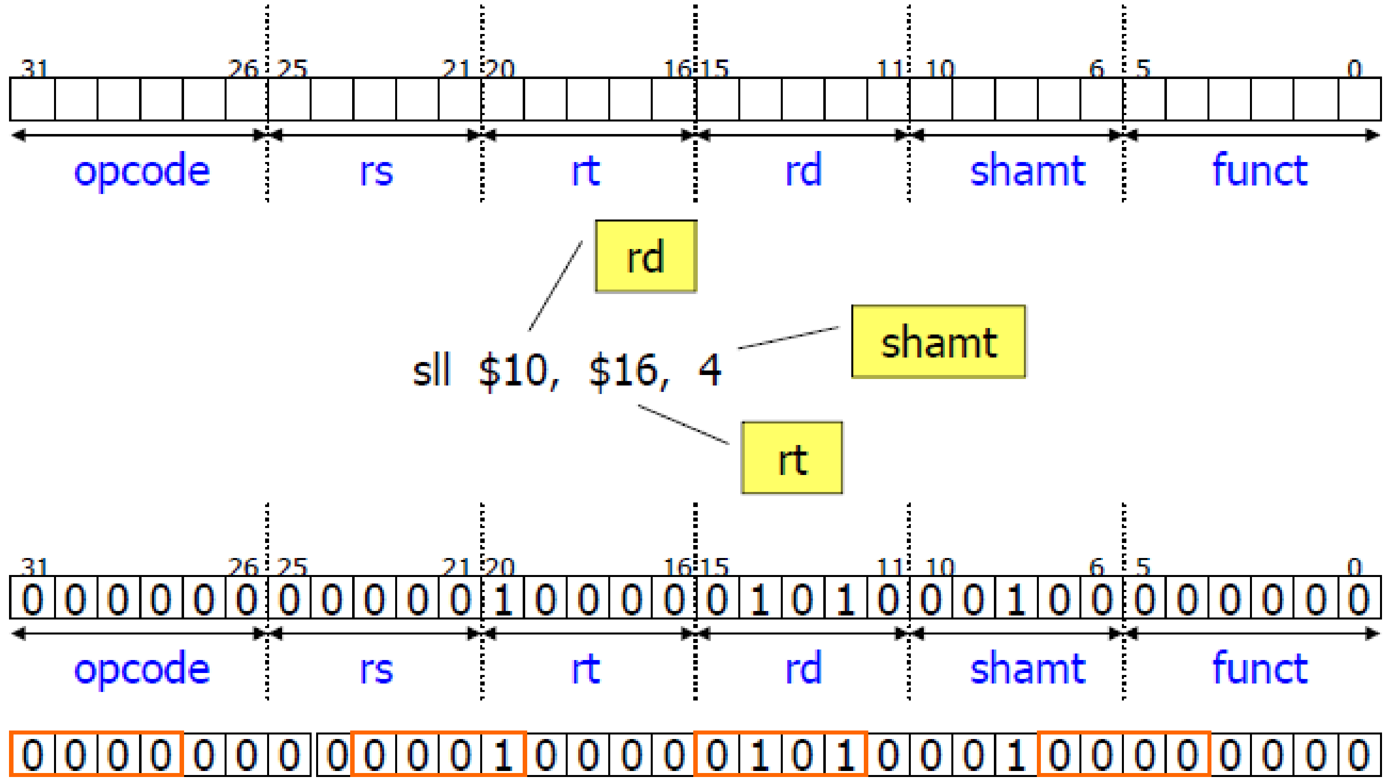
Encoding = 0x00622020

R-Type Encoding (sub)



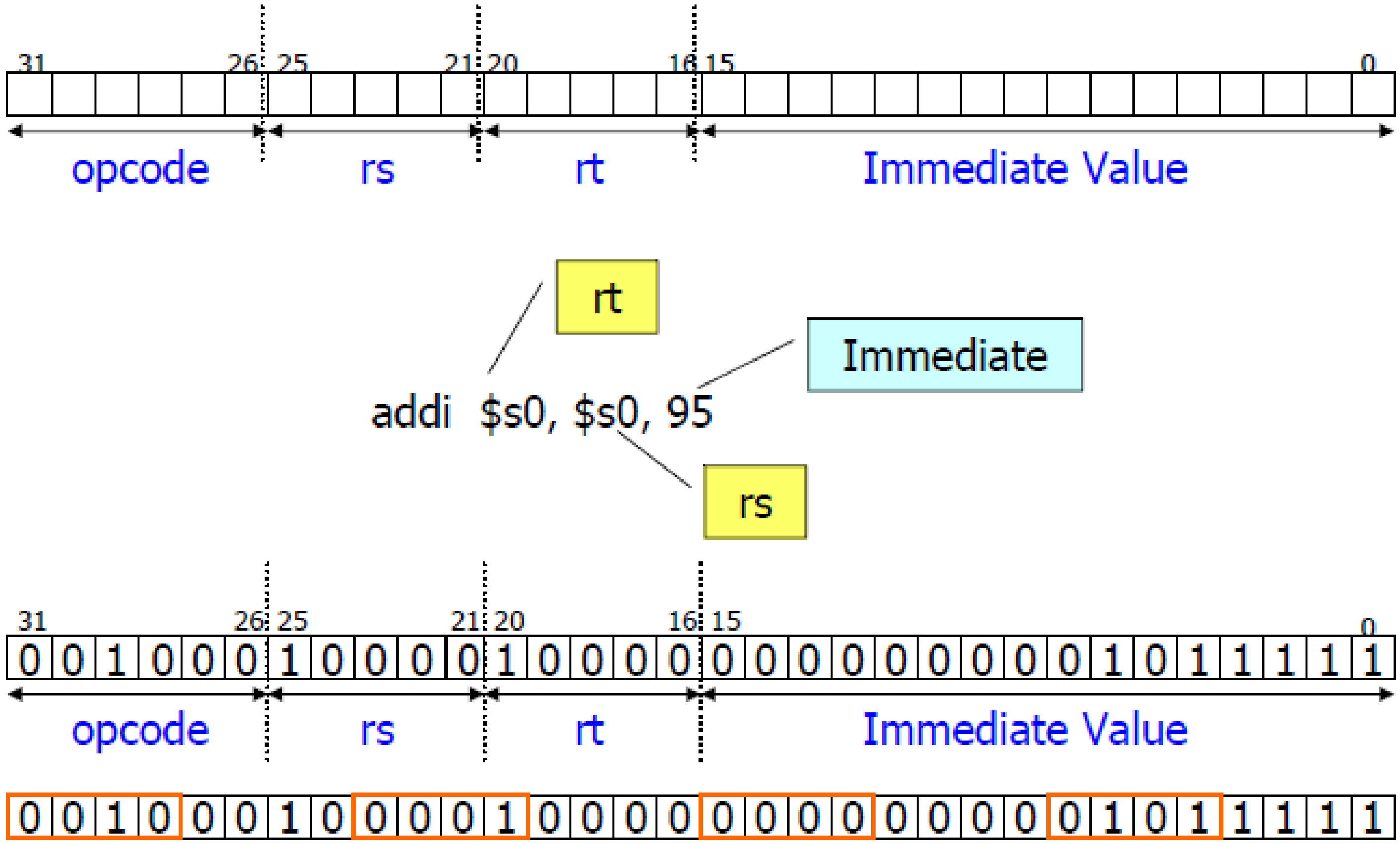
Encoding = 0x01098022

R-Type Encoding (sll)



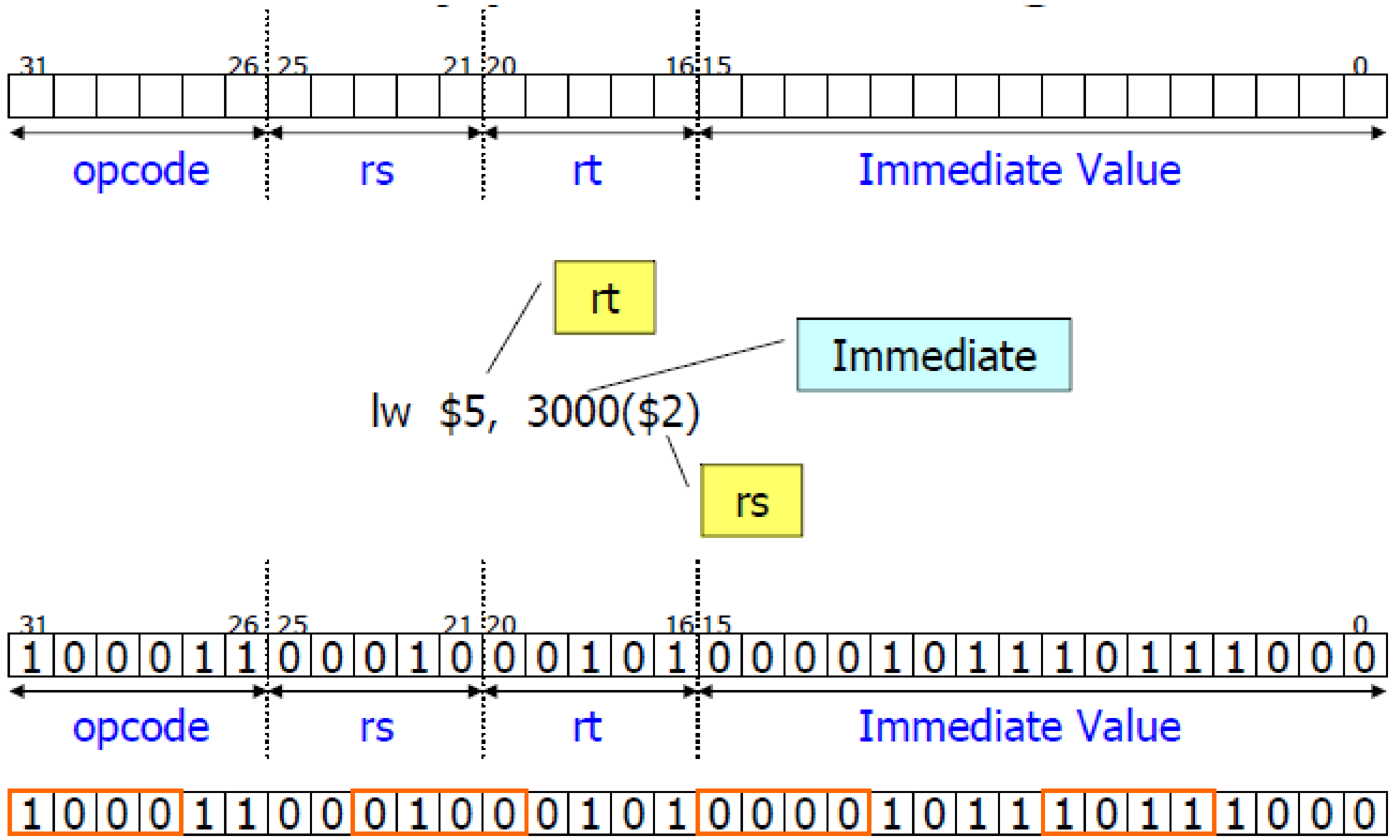
Encoding = 0x00105100

I-Type Encoding (addi)



Encoding = 0x2210005F

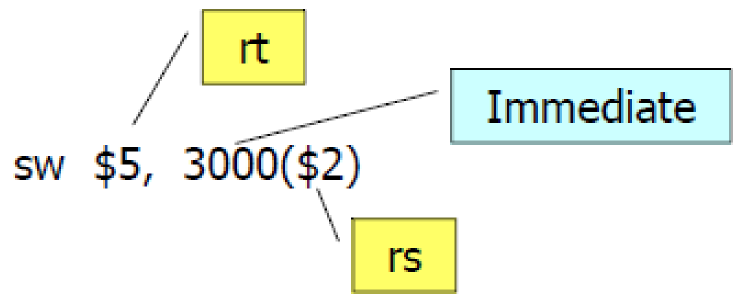
I-Type Encoding (lw)



Encoding = 0x8C450BB8

I-Type Encoding (sw)

31–26	25–21	20–16	15–0
opcode	rs	rt	Immediate Value



31–26	25–21	20–16	15–0
1 0 1 0 1 1	0 0 0 1 0	0 0 1 0 1	0 0 0 0 1 0 1 1 1 0 1 1 1 0 0 0
opcode	rs	rt	Immediate Value

1 0 1 0	1 1 0 0	0 1 0 0	0 1 0 1	0 0 0 0	1 0 1 1	1 0 1 1	1 0 0 0

Encoding = 0xAC450BB8

I-Type Encoding (sign extension)

- Immediate Field
  - Sign extended to 32-bits in
    - addi, lw/sw, andi, …
  - Example
    - **`addi $21,$22,-50`**

8	22	21	-50

001000	10110	10101	1111111111001110

Encoding Branch Instructions

- How to Encode Branch Instructions?
  - First figure out value for associated label
    - Will be done by assembler
  - Note MIPS has alignment restriction
    - All labels will be a multiple of 4
  - Label addresses divided by 4
    - Addresses encoded in terms of words
    - To increase address range

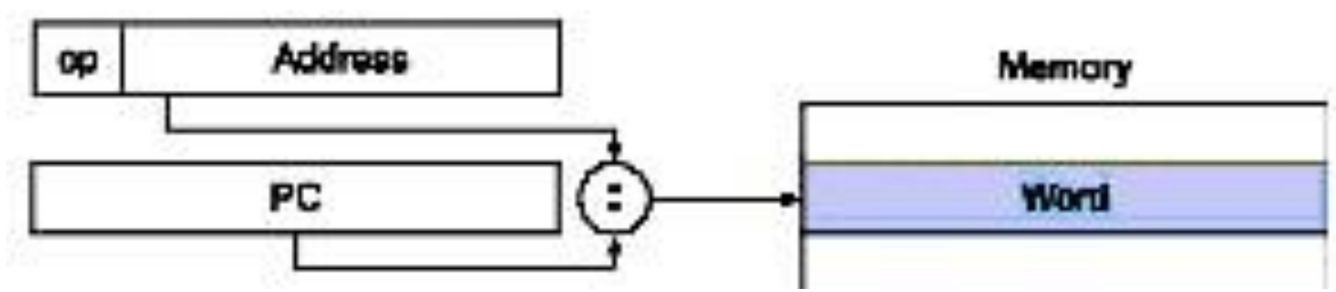
J-Type Instruction Encoding

`j target`	2	target
`jal target`	3	target
	6	26

- J-Type: jump and jump-and-link instructions
- 26 Bits for Target Field
  - Represents # of instructions instead of # of bytes
    - Represents 28 bits in terms of bytes
  - But PC requires 32 bits!!!
- Where do we get other 4 bits?

Pseudo-Direct Addressing

- Jump address is formed by:
  - Upper 4 bits of current PC
  - 26 bits of target address in instruction
  - Two bits of 0's



Questions

- Maximum code when using “j” instruction?
- Is there anyway to jump to a 32-bit address?

Questions

- Maximum code when using “j” instruction?
  - 256MB
- Is there anyway to jump to a 32-bit address?
  - use jr

Encoding jr instruction

- jump register (jr)
  - Unconditionally jump to address given by rs
  - After execution of jr $s0
    - PC = $s0
  - R-type, J-type, or I-type?

Encoding jr instruction

- jump register (jr)
  - Unconditionally jump to address given by rs
  - After execution of jr $s0
    - PC = $s0
  - R-type, J-type, or I-type?

jr rs

0	rs	0	8
6	5	15	6

Encoding Conditional Branch

- Where to Branch?
  - Branch # of instructions specified by offset
    - If rs equals to rt
  - Register holding address of current instruction
    - Program Counter (PC) / Instruction Register (IR)
  - What is value of PC after executing current instruction?

```
beq rs, rt, label
```

4	rs	rt	Offset
6	5	5	16

Encoding Conditional Branch (cont.)

- PC-Relative Addressing
  - Offset of conditional branch instructions relative to PC + 4
  - MIPS instructions are 4 bytes long → offset refers to number of words to next instruction instead of number of bytes

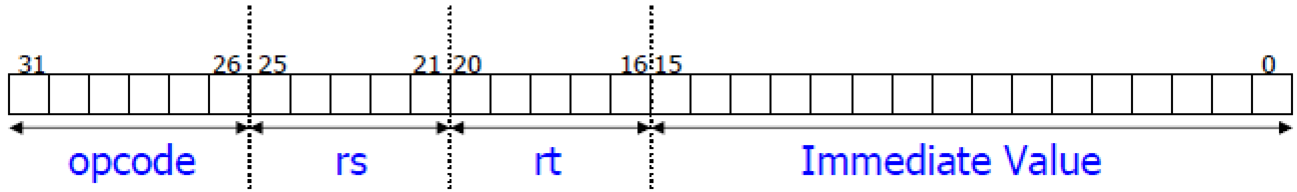
Encoding Conditional Branch (cont.)

- Branch Calculation
  - If we don't take branch:

    PC = PC+4 : byte # of next instruction
  - If we do take branch:

    PC = (PC+4) + (`immediate`*4)
  - `immediate` can be positive or negative

I-Type Encoding (bne)



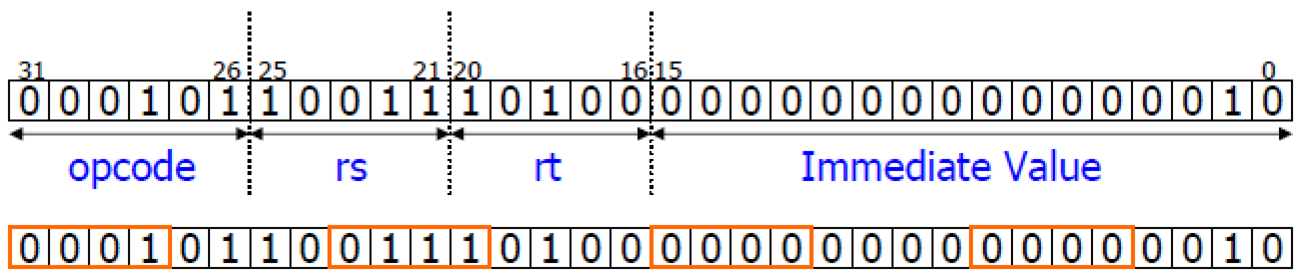

```
        bne $s3,$s4,Else:
        add $s0, $s1, $s2
        j     Exit:
Else:
        sub $s0, $s1, $s2
Exit:
```

rs

Label/Offset

bne $19, $20, Else

rt



Encoding = 0x16740002

Jump Instructions Summary

- Unconditional Jump (j, jal): Pseudo-direct addressing

`j target`

2	target
6	26

- Conditional Jump (bne, beq) : PC-relative addressing

`beq rs, rt, label`

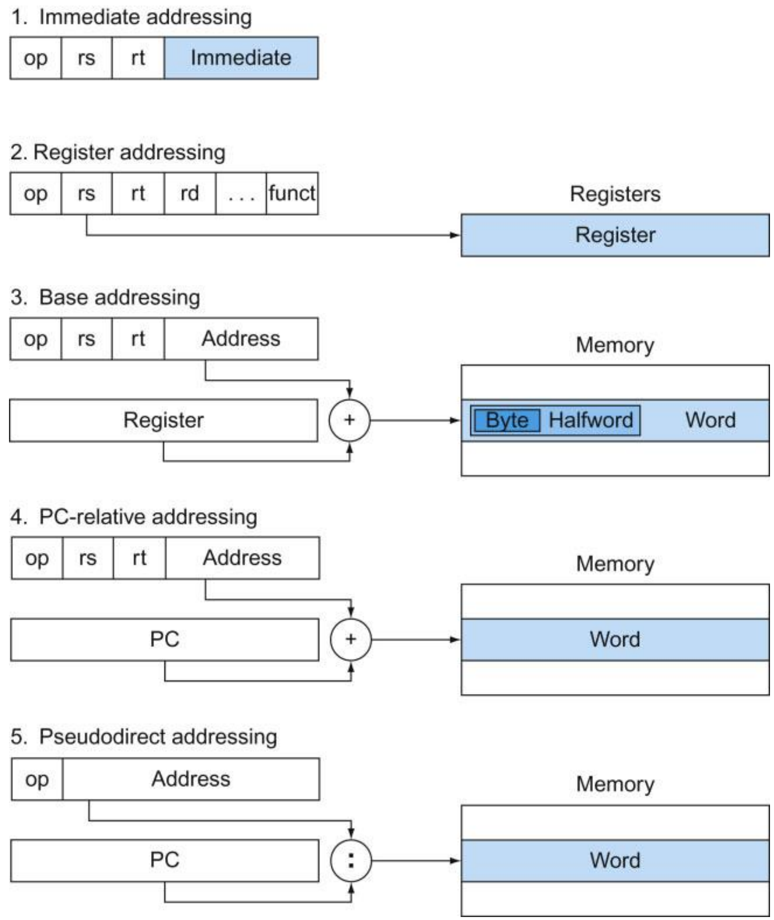
4	rs	rt	Offset
6	5	5	16

- Jump Register (jr): Register addressing

`jr rs`

0	rs	0	8
6	5	15	6

Addressing Mode Summary



Contents

Supplementary Tips

- Big Immediate
- Byte/ Halfword Operations
- More Shift Instructions
- Endianness

Big Immediate

- In MIPS, immediate field has 16 bits
- For 32-bit immediate operands
  - MIPS includes load upper immediate (lui)
    - Which sets upper 16 bits of a constant in a register and fills lower 16 bits with 0's
  - Then one can set lower 16 bits using ori

`lui $s0,61`	0000 0000 0111 1101	0000 0000 0000 0000
`ori $s0,$s0,2304`	0000 0000 0111 1101	0000 1001 0000 0000

Byte/ Halfword Operations

- Load byte/halfword (sign extend to 32 bits in rt)
  - `lb rt,offset(rs)`
  - `lh rt,offset(rs)`
- Load unsigned byte/halfword (zero extend to 32 bits in rt)
  - `lbu rt,offset(rs)`
  - `lhu rt,offset(rs)`
- Store just rightmost byte/halfword
  - `sb rt,offset(rs)`
  - `sh rt,offset(rs)`

Shift Instructions

- Logical shift left/right (by specified number of bits)
  - `sll rd,rt,shamt`
  - `srl rd,rt,shamt`
- Logical shift left/right (by variable number of bits)
  - `sllv rd,rt,rs`
  - `srlv rd,rt,rs`
- Arithmetic Shift (keep sign while shifting)
  - `sra rd,rt,shamt`
  - `srav rd,rt,rs`

Big Endian vs. Little Endian

- How a multiple byte data word stored in memory
- Big Endian
  - Most significant byte of a multi-byte word is stored at lowest memory address (e.g. Sun Sparc, PowerPC)
- Little Endian
  - Least significant byte of a multi-byte word is stored at lowest memory address (e.g. Intel x86)
  - LLL (Least significant in Lowest address)

BIG ENDIAN

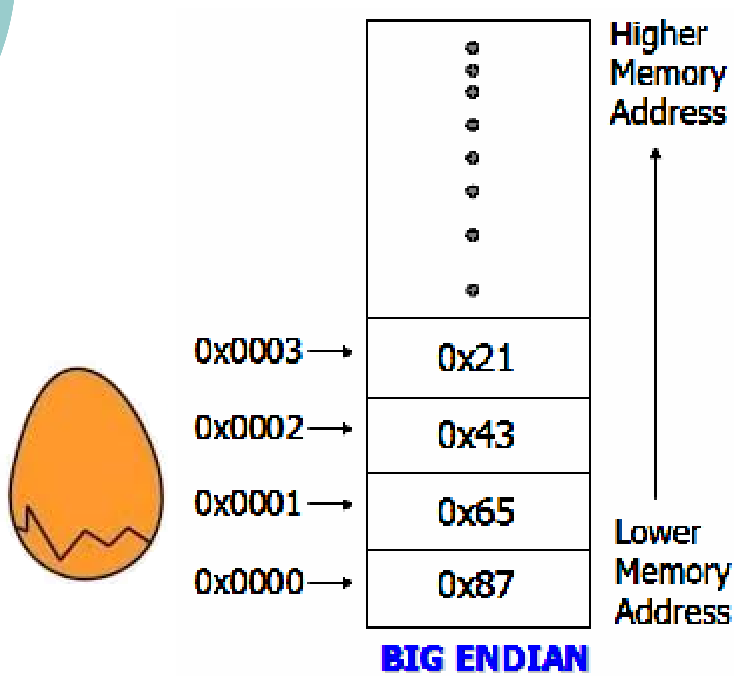
The way people always broke their eggs in the Lilliput land

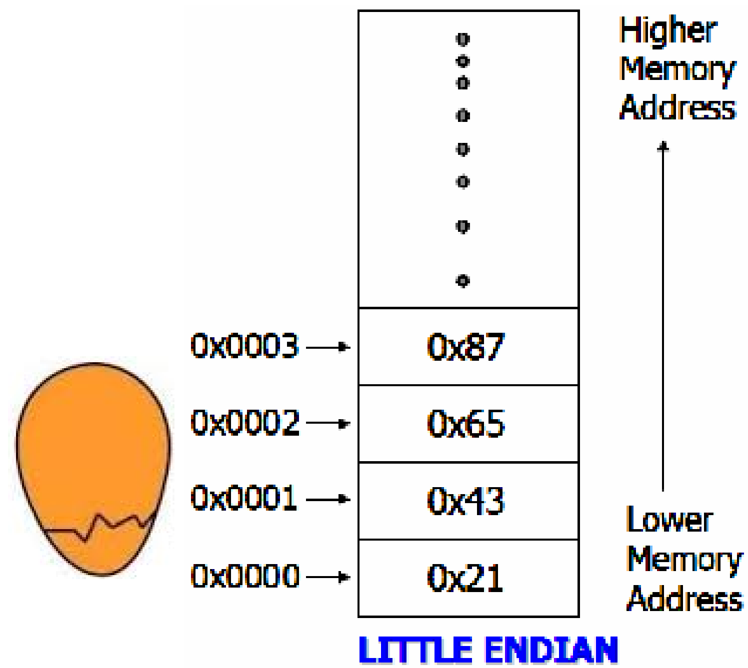
LITTLE ENDIAN

The way the king then ordered the people to break their eggs

Example of Endianness

Store 0x87654321 at address 0x0000, byte-addressable





Why Worry?

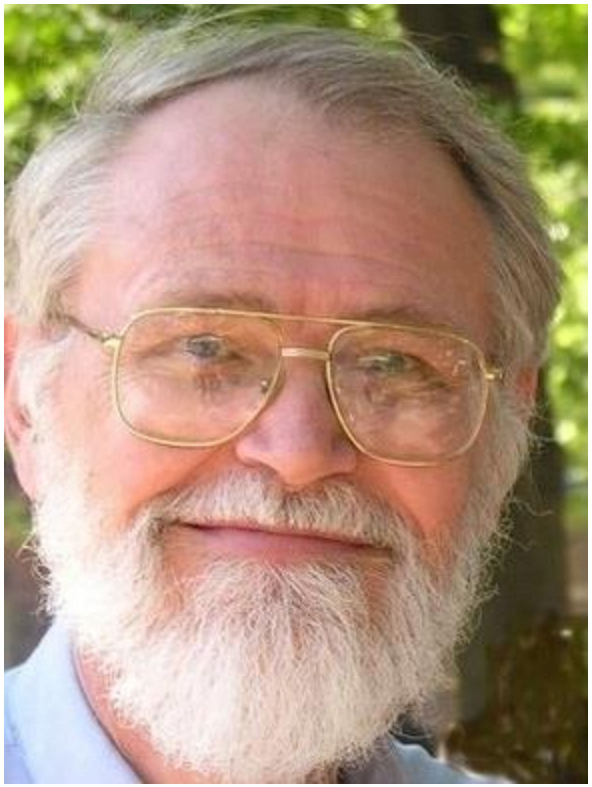
- Two computers with different byte orders may be communicating
- Failure to account for varying endianness → hard to detect bug
- Read a 32-bit value and store it in a 32-bit register
  - Do I need to know Endianness? (No)
- Endianness only makes sense when you are breaking up a multi-byte quantity

MIPS Registers (revisiting)

Register Number	Mnemonic Name	Conventional Use		Register Number	Mnemonic Name	Conventional Use
$0	$zero	Permanently 0		$24, $25	$t8, $t9	Temporary
$1	$at	Assembler Temporary (reserved)		$26, $27	$k0, $k1	Kernel (reserved for OS)
$2, $3	$v0, $v1	Value returned by a subroutine		$28	$gp	Global Pointer
$4–$7	$a0–$a3	Arguments to a subroutine		$29	$sp	Stack Pointer
$8–$15	$t0–$t7	Temporary (not preserved across a function call)		$30	$fp	Frame Pointer
$16–$23	$s0–$s7	Saved registers (preserved across a function call)		$31	$ra	Return Address

MIPS Processor Summary

- MIPS Registers
- MIPS Instructions
  - Arithmetic (add, sub, mul, div)
  - Data Transfer (load & store)
  - Logical & Shift
  - Decision Making (conditional/ unconditional)
- Subroutine Call/ Return
- MIPS Instruction Encoding
- MIPS Addressing Modes
- Other issues:
  - Endianness, Big Immediate, Byte/Halfword Ops, System Calls



Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition not smart enough to debug it.

Brian Karnighan