Computer Architecture

Chapter 2: Instruction – Language of the Computer

Assoc. Prof. Dinh Duc Anh Vu

Adapted from Computer Organization the Hardware/Software Interface – 5th

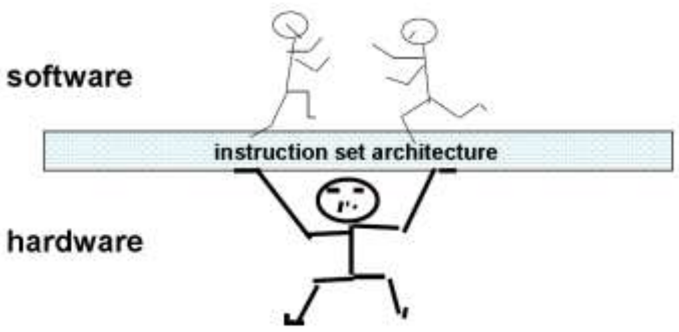
Chapter 2: MISP - ISA

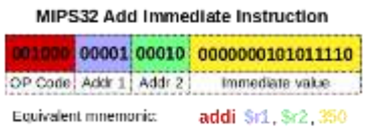
Introduction

- **Language**: a system of communication consisting of sounds, words, and grammar, or the system of communication used by people in a particular country or type of work (Oxford Dictionary)
- To command a computer's hardware: speak its language

http://media.apnarm.net.au/img/media/images/2013/08/14/computer_language_t620.jpg 2

Instruction Set Architecture (ISA)

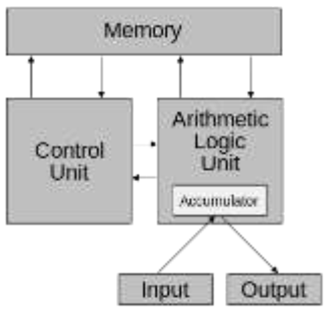




3

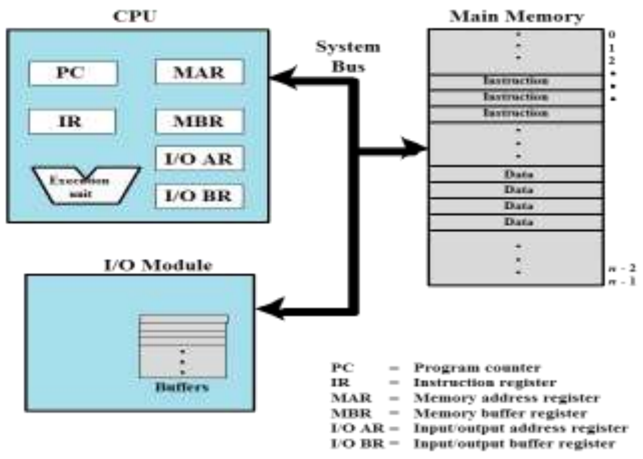
Von Neumann Architecture

- Stored-program concept
- Instruction category:
  - Arithmetic
  - Data transfer
  - Logical
  - Conditional branch
  - Unconditional jump

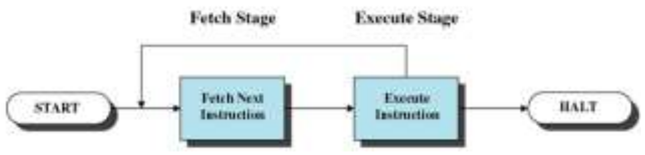


4

Computer Components



Instruction Execution



- Instruction fetch: from the memory
  - PC increased
  - PC stores the next instruction
- Execution: decode and execute

The MIPS Instruction Set

- MIPS architecture
- MIPS Assembly Inst. ⬄ MIPS Machine Instr.
- Assembly:
  - `add $t0, $s2, $t0`
- Machine:
  - 000000_10010_01000_01000_00000_100000
- Only **<u>one operation</u>** is performed per MIPS instruction

IS Design Principles

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

MIPS Operands

- 32 32-bit registers
  - `$s0-$s7`: corresponding to variables
  - `$t0-$t9`: storing temporary value
  - `$a0-$a3`
  - `$v0-$v1`
  - `$gp, $fp, $sp, $ra, $at, $zero, $k0-$k1`
- 2^{30} memory words (4 byte): accessed only by data transfer instructions (memory operand)
- Immediate

Arithmetic Instructions

Opcode	Destination register	Source register 1	Source register 2(*)

- Opcode:
  - `add`: DR = SR1 + SR2
  - `sub`: DR = SR1 – SR2
  - `addi`: SR2 is an immediate (e.g. 20), DR = SR1 + SR2
- Three register operands

Design Principle 1

- **Simplicity favours regularity**
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

Arithmetic Instructions: Example

- **Q**: what is MIPS code for the following C code

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

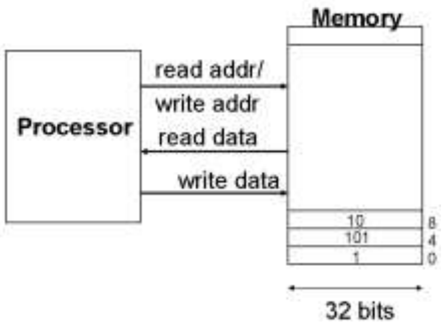
If the variables `g`, `h`, `i`, `j`, and `f` are assigned to the register `$s0, $s1, $s2, $s3`, and `$s4`, respectively.

- **A**:

```
add $t0, $s0, $s1 # g + h
add $t1, $s2, $s3 # i + j
sub $s4, $t0, $t1 # t0 - t1
```

Data Transfer Instructions

- Move data b/w memory and registers
  - Register
  - Address: a value used to delineate the location of a specific data element within a memory array
- Load: **<u>copy</u>** data from memory to a register
- Store: **<u>copy</u>** data from a register to memory



Data Transfer Instructions (cont.)

Opcode	Register	Memory address

- Memory address: *offset(base register)*
  - Byte address: each address identifies an 8-bit byte
  - “words” are aligned in memory (address must be multiple of 4)

Data Transfer Instructions (cont.)

- Opcode:
  - lw: load word
  - sw: store word
  - lh: load half ($s1 = {16{M[$s2+imm][15]},M[$s2 + imm]})
  - lhu: load half unsigned ($s1 = {16'b0,M[$s2 + imm]})
  - sh: store half
  - lb: load byte
  - lbu: load byte unsigned
  - sb: store byte
  - ll: load linked word
  - sc: store conditional
  - lui: load upper immediate $s1 = {imm,16'b0}

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 Operand Example 1

- C code:

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

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

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

Memory Operand Example 2

- C code:

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

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

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

Exercise

- Show the effects on memory and registers of the following instructions. Suppose a portion of memory contains the following data

address 0x10000000 0x10000003

0x12	0x34	0x56	0x78
0x9A	0xBC	0xDE	0xF0

address 0x10000004 0x10000007

- And register $t0 contains 0x10000000 and $s0 contains 0x01234567. Assume each of the following instructions is executed independently of the others, starting with the values given above

- a) lw $t1, 0($t0)
- b) lw $t2, 4($t0)
- c) lb $t3, 0($t0)
- d) lb $t4, 4($t0)
- e) lb $t5, 3($t0)
- f) lh $t6, 4($t0)
- g) sw $s0, 0($t0)
- h) sb $s0, 4($t0)
- i) sb $s0, 7($t0)

Exercise

- Convert the following C statements to equivalent MIPS assembly language if the variables f, g, and h are assigned to registers $s0, $s1, and $s2 respectively. Assume that the base address of the array A and B are in registers $s6 and $s7, respectively.
- 1) f = g + h + B[4]
- 2) f = g – A[B[4]]

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!

Immediate Operands

- Constant data specified in an instruction
  `addi $s3, $s3, 4`
- No subtract immediate instruction
  - Just use a negative constant
    `addi $s2, $s1, -1`
- *Design Principle 3:* Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction

The Constant Zero

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

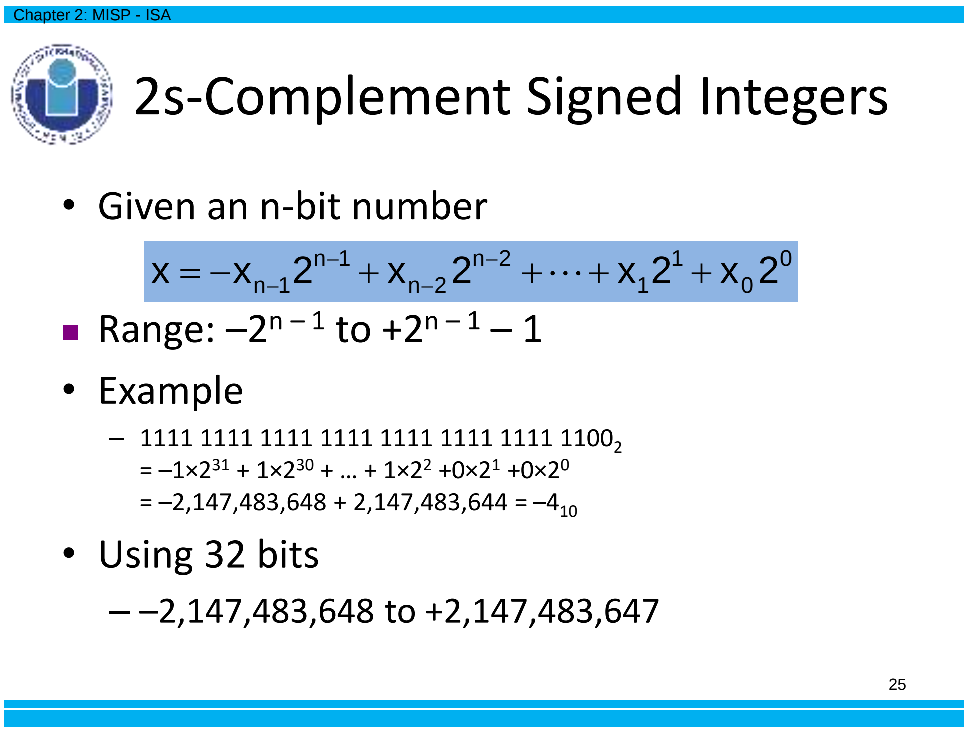
    ```
    add $t2, $s1, $zero
    ```

Unsigned Binary Integers

- 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

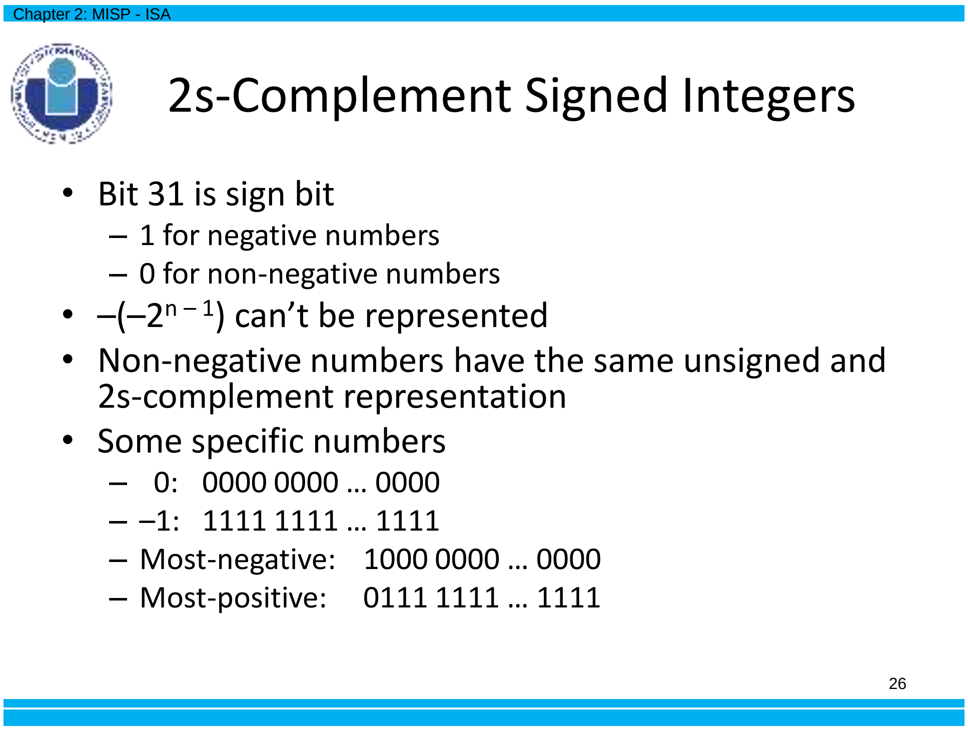


2s-Complement Signed Integers

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



2s-Complement Signed Integers

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

Signed Negation

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

$$x + \bar{x} = 1111\ldots111_2 = -1$$

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

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

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 R-format Instructions

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

- Instruction fields
  - op: operation code (opcode)
  - rs: first source register number
  - rt: second source register number
  - rd: destination register number
  - shamt: shift amount (00000 for now)
  - funct: function code (extends opcode)

R-format Example

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

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

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

$00000010001100100100000000100000_2 = 02324020_{16}$

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
  - rt: destination or source register number
  - Constant: -2^{15} to $+2^{15} - 1$
  - Address: offset added to base address in rs
- Example: `lw $t0, 32($s3)`

35_d	19_d	8	32
opcode	\$s3	\$t0	address

Design Principle

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

MIPS Instructions Format Summary

Instr.	Type	op	rs	rt	rd	shamt	function	address
add	R	0	reg	reg	reg	0	32_d	n.a.
sub	R	0	reg	reg	reg	0	34_d	n.a.
addi	I	8_d	reg	reg	n.a.	n.a.	n.a.	constant
lw	I	35_d	reg	reg	n.a.	n.a.	n.a.	address
sw	I	43_d	reg	reg	n.a.	n.a.	n.a.	address

- R-format: arithmetic instructions
- I-format: data transfer instructions

Example

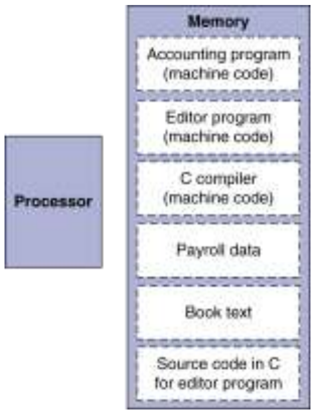
- Write MIPS code for the following C code, then translate the MIPS code to machine code

```
A[300] = h + A[300] - 2;
```

- Assume that `$t1` stores the base of array A and `$s2` stores h

Stored Program Computers

The BIG Picture


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

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

Logical Operations

- Instructions for bitwise manipulation

Operation	C	Java	MIPS
Shift left	<<	<<	`sll`
Shift right	>>	>>>	`srl`
Bitwise AND	&	&	`and, andi`
Bitwise OR	|	|	`or, ori`
Bitwise NOT	~	~	`nor`

- Useful for extracting and inserting groups of bits in a word

Shift Operations

- ***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)
- Example: `sll $t2, $s0, 4 # $t2 = $s0 << 4`

0	0	16	10	4	0
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

AND Operations

- Useful to mask bits in a word
  - Select some bits, clear others to 0

`and $t0, $t1, $t2`

$t2	0000 0000 0000 0000 0000 1101 1100 0000
$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	0000 0000 0000 0000 0000 1100 0000 0000

OR Operations

- Useful to include bits in a word
  - Set some bits to 1, leave others unchanged

```
or $t0, $t1, $t2
```

$t2	0000 0000 0000 0000 0000 1101 1100 0000
$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	0000 0000 0000 0000 0011 1101 1100 0000

NOT Operations

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

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

Register 0: always read as zero

$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	1111 1111 1111 1111 1100 0011 1111 1111

Example

- The data table below contains the values for register $t0 and $t1

a	**$t0 = 0xAAAAAAAA, $t1 = 0x12345678**
b	$t0 = 0xF00DD00D, $t1 = 0x11111111

- Find the value for $t2 after the following sequence of instruction?

```
sll $t2, $t0, 44        sll $t2, $t0, 4         srl $t2, $t0, 3
or $t2, $t2, $t1        andi $t2, $t2, -2       andi $t2, $t2, 0xFFEF
```

Conditional Operations

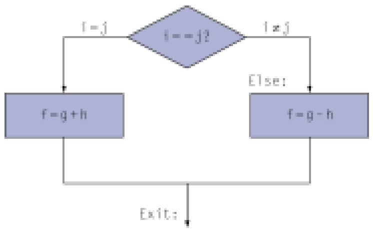
- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- `beq rs, rt, L1`
  - if (rs == rt) branch to instruction labeled L1;
- `bne rs, rt, L1`
  - if (rs != rt) branch to instruction labeled L1;
- `j L1`
  - unconditional jump to instruction labeled L1

Compiling If Statements

- C code:

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

  – f, g, … in $s0, $s1, …



- Compiled MIPS code:

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

Assembler calculates addresses

Compiling Loop Statements

- C code:

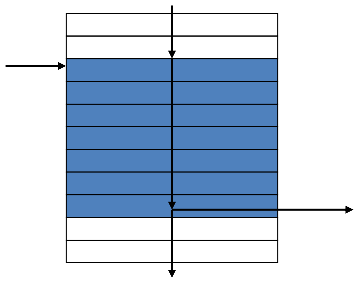
```
while (save[i] == k) i += 1;
```

  – i in $s3, k in $s5, base address of save in $s6
- Compiled MIPS code:

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

Basic Blocks

- A basic block is a sequence of instructions with
  - – No embedded branches (except at end)
  - – No branch targets (except at beginning)



- A compiler identifies basic blocks for optimization
- An advanced processor can accelerate execution of basic blocks

More Conditional Operations

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

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

Branch Instruction Design

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

Signed vs. Unsigned

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

Procedure Calling

- Steps required
  - 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

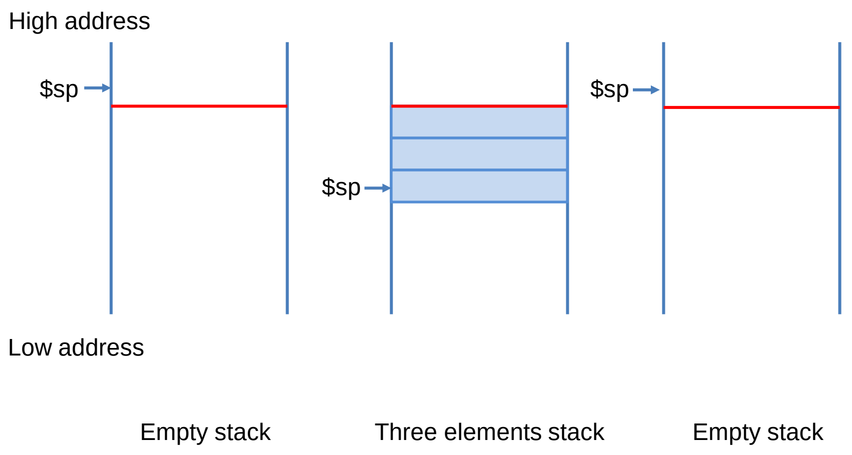
Register Usage

- $a0 – $a3: arguments (reg’s 4 – 7)
- $v0, $v1: result values (reg’s 2 and 3)
- $t0 – $t9: temporaries
  - Can be overwritten by callee
- $s0 – $s7: saved
  - Must be saved/restored by callee
- $gp: global pointer for static data (reg 28)
- $sp: stack pointer (reg 29)
- $fp: frame pointer (reg 30)
- $ra: return address (reg 31)

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

Stack Address Model



Leaf Procedure Example

- C code:

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

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

Leaf Procedure Example

- MIPS code:

`leaf_example:`	

`  addi $sp, $sp, -4`
`  sw   $s0, 0($sp)`
	Save $s0 on stack

`  add  $t0, $a0, $a1`
`  add  $t1, $a2, $a3`
`  sub  $s0, $t0, $t1`
	Procedure body

`  add  $v0, $s0, $zero`	Result

`  lw   $s0, 0($sp)`
`  addi $sp, $sp, 4`
	Restore $s0

`  jr   $ra`	Return

Non-Leaf Procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
  - Its return address
  - Any arguments and temporaries needed after the call
- Restore from the stack after the call

Non-Leaf Procedure Example

- C code:

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

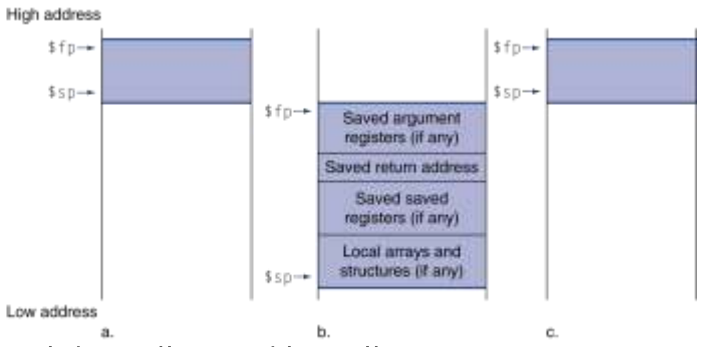
  - Argument n in $a0
  - Result in $v0

Non-Leaf Procedure Example

- MIPS code:

```
fact:
    addi $sp, $sp, -8     # adjust stack for 2 items
    sw   $ra, 4($sp)      # save return address
    sw   $a0, 0($sp)      # save argument
    slti $t0, $a0, 1      # test for n < 1
    beq  $t0, $zero, L1
    addi $v0, $zero, 1    # if so, result is 1
    addi $sp, $sp, 8      #   pop 2 items from stack
    jr   $ra              #   and return
L1: addi $a0, $a0, -1     # else decrement n
    jal  fact             # recursive call
    lw   $a0, 0($sp)      # restore original n
    lw   $ra, 4($sp)      #   and return address
    addi $sp, $sp, 8      # pop 2 items from stack
    mul  $v0, $a0, $v0    # multiply to get result
    jr   $ra              # and return
```

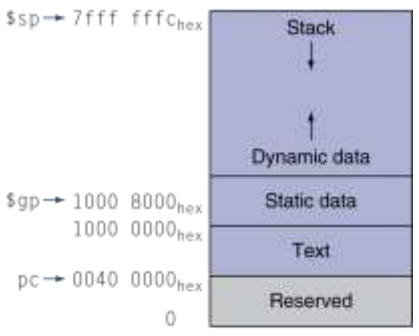
Local Data on the Stack


High address
$fp
$sp
$fp
Saved argument registers (if any)
Saved return address
Saved saved registers (if any)
Local arrays and structures (if any)
$sp
$fp
$sp
Low address
a.
b.
c.

- 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, new in Java
- Stack: automatic storage


$sp → 7fff fffc$_{hex}$
$gp → 1000 8000$_{hex}$
1000 0000$_{hex}$
pc → 0040 0000$_{hex}$
0
Stack
Dynamic data
Static data
Text
Reserved