香 港 中 文 大 學
The Chinese University of Hong Kong

CENG3420
L03: Instruction Set Architecture

Bei Yu

byu@cse.cuhk.edu.hk

(Latest update: January 31, 2018)

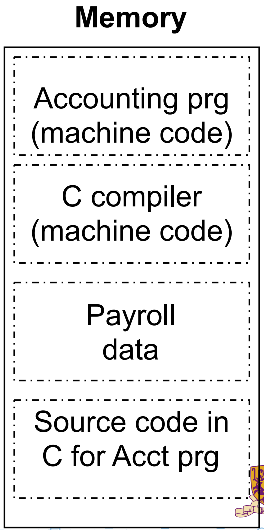
Spring 2018

Overview

Overview

Two Key Principles of Machine Design

1. Instructions are represented as numbers and, as such, are indistinguishable from data
2. Programs are stored in alterable memory (that can be read or written to) just like data


Memory
Accounting prg (machine code)
C compiler (machine code)
Payroll data
Source code in C for Acct prg

Stored-Program Concept

- Programs can be shipped as files of binary numbers – binary compatibility
- Computers can inherit ready-made software provided they are compatible with an existing ISA – leads industry to align around a small number of ISAs

Assembly Language Instructions

The language of the machine

- Want an ISA that makes it easy to build the hardware and the compiler while maximizing performance and minimizing cost

Our target: the MIPS ISA

- similar to other ISAs developed since the 1980's
- used by Broadcom, Cisco, NEC, Nintendo, Sony, ...

Design Goals

Maximize performance, minimize cost, reduce design time (time-to-market), minimize memory space (embedded systems), minimize power consumption (mobile systems)

CISC vs. RISC

Complex Instruction Set Computer (CISC)

Lots of instructions of variable size, very memory optimal, typically less registers.

- Intel x86

Reduced Instruction Set Computer (RISC)

Instructions, all of a fixed size, more registers, optimized for speed. Usually called a "Load/Store" architecture.

- MIPS, LC-3b, Sun SPARC, HP PA-RISC, IBM PowerPC ...

RISC – Reduced Instruction Set Computer

RISC Philosophy

- fixed instruction lengths
- load-store instruction sets
- limited number of addressing modes
- limited number of operations

- Instruction sets are measured by how well compilers use them as opposed to how well assembly language programmers use them

MIPS (RISC) Design Principles

Simplicity favors regularity

- fixed size instructions
- small number of instruction formats
- opcode always the first 6 bits

Smaller is faster

- limited instruction set
- limited number of registers in register file
- limited number of addressing modes

Make the common case fast

- arithmetic operands from the register file (load-store machine)
- allow instructions to contain immediate operands

Good design demands good compromises

- three instruction formats

MIPS Instruction Fields

MIPS fields are given names to make them easier to refer to

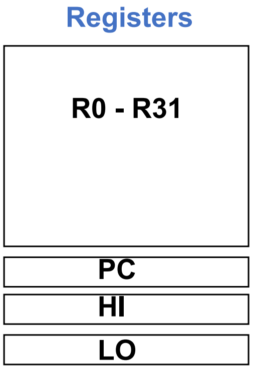
6	5	5	5	5	6
op	rs	rt	rd	shamt	funct

op	6-bits,	opcode that specifies the operation
rs	5-bits,	register file address of the first source operand
rt	5-bits,	register file address of the second source operand
rd	5-bits,	register file address of the result’s destination
shamt	5-bits,	shift amount (for shift instructions)
funct	6-bits,	function code augmenting the opcode

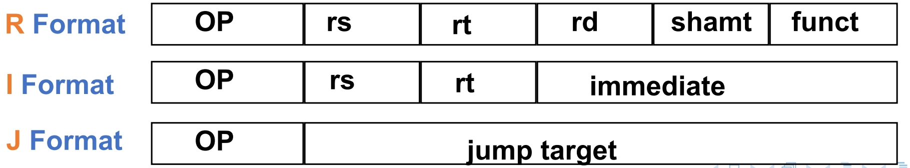
The MIPS ISA

Instruction Categories

- Load/Store
- Computational
- Jump and Branch
- Floating Point
- Memory Management
- Special



3 Instruction Formats: all 32 bits wide

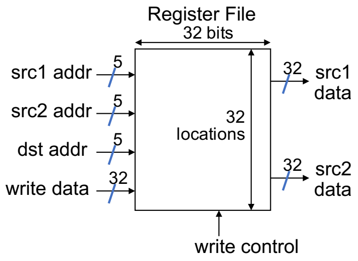


MIPS Instruction Classes Distribution

Frequency of MIPS instruction classes for SPEC2006

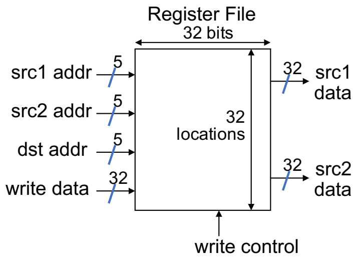
Instruction Class	**Frequency**	
	Integer	Ft. Pt.
Arithmetic	16%	48%
Data transfer	35%	36%
Logical	12%	4%
Cond. Branch	34%	8%
Jump	2%	0%

MIPS Register File



- Holds thirty-two 32-bit registers
- Two read ports
- One write port

MIPS Register File



- Holds thirty-two 32-bit registers
- Two read ports
- One write port

Registers are

- Faster than main memory
  - But register files with more locations are slower
  - E.g., a 64 word file may be 50% slower than a 32 word file
  - Read/write port increase impacts speed quadratically
- Easier for a compiler to use
  - `(A*B)-(C*D)-(E*F)` can do multiplies in any order vs. stack
- Can hold variables so that code density improves (since register are named with fewer bits than a memory location)

Aside: MIPS Register Convention

Name	Register Number	Usage	Preserve on call?
$zero	0	constant 0 (hardware)	n.a.
$at	1	reserved for assembler	no
$v0 - $v1	2-3	returned values	no
$a0 - $a3	4-7	arguments	no
$t0 - $t7	8-15	temporaries	no
$s0 - $s7	16-23	saved values	yes
$t8 - $t9	24-25	temporaries	no
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return addr (hardware)	yes

History of MIPS

1981	Dr. John Hennessy at Stanford University founds and leads Stanford MIPS
1984	MIPS Computer Systems, Inc.
1986	R2000 microprocessor
1988	R3000 microprocessor
1991	R4000 microprocessor
1992	Acquired by SGI, rename to MIPS Technologies, Inc
1994	R8000 microprocessor
2011	Android-MIPS
2011	Sold to Imagination Technologies
Sep., 2017	Sold to Tallwood Venture Capital as Tallwood MIPS Inc. for $65 million

History of MIPS (cont.)

- Used in many embedded systems
- E.g., Nintendo-64, Playstation 1, Playstation 2

Overview

MIPS Arithmetic Instructions

- MIPS assembly language arithmetic statement

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

- Each arithmetic instruction performs one operation
- Each specifies exactly three operands that are all contained in the datapath's register file (`$t0,$s1,$s2`)

```
destination  = source1 op source2
```

- Instruction Format (R format)

0	17	18	8	0	0x22

MIPS Immediate Instructions

- Small constants are used often in typical code

Possible approaches?

- put “typical constants” in memory and load them
- create hard-wired registers (like `$zero`) for constants like 1
- have special instructions that contain constants

```
addi $sp, $sp, 4       #$sp = $sp + 4
slti $t0, $s2, 15      #$t0 = 1 if $s2<15
```

MIPS Immediate Instructions

- Small constants are used often in typical code

Possible approaches?

- put "typical constants" in memory and load them
- create hard-wired registers (like `$zero`) for constants like 1
- have special instructions that contain constants

```
addi $sp, $sp, 4       #$sp = $sp + 4
slti $t0, $s2, 15      #$t0 = 1 if $s2<15
```

- Machine format (I format)
- The constant is kept inside the instruction itself!
- Immediate format limits values to the range -2^{15} to $+2^{15}-1$

Aside: How About Larger Constants?

- We'd also like to be able to load a 32 bit constant into a register
- For this we must use two instructions

1. A new "load upper immediate" instruction

```
lui $t0, 1010101010101010
```

2. Then must get the lower order bits right, use

```
ori $t0, $t0, 1010101010101010
```

Aside: How About Larger Constants?

- We'd also like to be able to load a 32 bit constant into a register
- For this we must use two instructions

1. A new "load upper immediate" instruction

```
lui $t0, 1010101010101010
```

2. Then must get the lower order bits right, use

```
ori $t0, $t0, 1010101010101010
```

1010101010101010	0000000000000000
0000000000000000	1010101010101010
1010101010101010	1010101010101010

MIPS Shift Operations

- Need operations to pack and unpack 8-bit characters into 32-bit words
- Shifts move all the bits in a word left or right

```
sll $t2, $s0, 8     #$t2 = $s0 << 8 bits
srl $t2, $s0, 8     #$t2 = $s0 >> 8 bits
```

- Instruction Format (R format)
- Such shifts are called logical because they fill with zeros
- Notice that a 5-bit shamt field is enough to shift a 32-bit value $2^5 - 1$ or 31 bit positions

MIPS Logical Operations

There are a number of bit-wise logical operations in the MIPS ISA

R Format

```
and $t0, $t1, $t2     #$t0 = $t1 & $t2
or  $t0, $t1, $t2     #$t0 = $t1 | $t2
nor $t0, $t1, $t2     #$t0 = not($t1 | $t2)
```

I Format

```
andi $t0, $t1, 0xFF00     #$t0 = $t1 & ff00
ori  $t0, $t1, 0xFF00     #$t0 = $t1 | ff00
```

Overview

MIPS Memory Access Instructions

- Two basic data transfer instructions for accessing memory

```
lw   $t0, 4($s3)   #load word from memory
sw   $t0, 8($s3)   #store word to memory
```

- The data is loaded into (`lw`) or stored from (`sw`) a register in the register file – a 5 bit address
- The memory address – a 32 bit address – is formed by adding the contents of the base address register to the offset value
- A 16-bit field meaning access is limited to memory locations within a region of $\pm 2^{13}$ or 8,192 words ($\pm 2^{15}$ or 32,768 bytes) of the address in the base register

Machine Language – Load Instruction

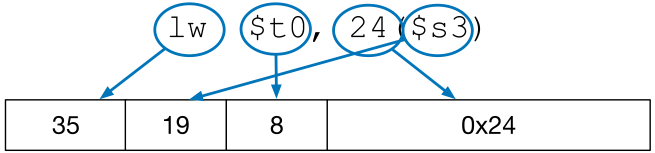
Load/Store Instruction Format (I format):

```
lw  $t0, 24($s3)
```

35	19	8	0x24

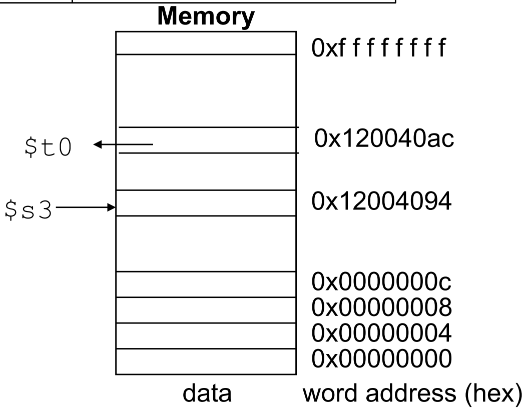
Machine Language – Load Instruction

Load/Store Instruction Format (I format):



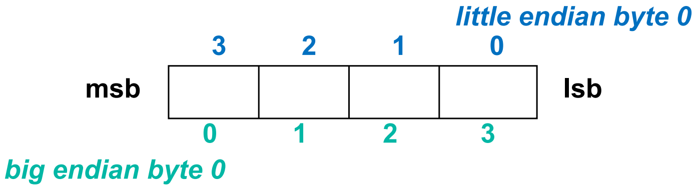
24_{10} + $s3 =

```
  . . . 0001 1000
+ . . . 1001 0100
  . . . 1010 1100 =
        0x120040ac
```



Byte Addresses

- Since 8-bit bytes are so useful, most architectures address individual bytes in memory
- Alignment restriction – the memory address of a word must be on natural word boundaries (a multiple of 4 in MIPS-32)
- Big Endian: leftmost byte is word address
  - IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- Little Endian: rightmost byte is word address
  - Intel 80x86, DEC Vax, DEC Alpha (Windows NT)



Aside: Loading and Storing Bytes

MIPS provides special instructions to move bytes

```
lb   $t0, 1($s3)   #load byte from memory
sb   $t0, 6($s3)   #store byte to  memory
```

- What 8 bits get loaded and stored?
- Load byte places the byte from memory in the rightmost 8 bits of the destination register
- Store byte takes the byte from the rightmost 8 bits of a register and writes it to a byte in memory

EX-1:

Given following code sequence and memory state:

```
add     $s3, $zero, $zero
lb      $t0, 1($s3)
sb      $t0, 6($s3)
```

Memory	
0x 0 0 0 0 0 0 0 0	24
0x 0 0 0 0 0 0 0 0	20
0x 0 0 0 0 0 0 0 0	16
0x 1 0 0 0 0 0 1 0	12
0x 0 1 0 0 0 4 0 2	8
0x F F F F F F F F	4
0x 0 0 9 0 1 2 A 0	0
Data	Word Address (Decimal)

1. What value is left in `$t0`?
2. What word is changed in Memory and to what?
3. What if the machine was little Endian?

Overview

MIPS Control Flow Instructions

MIPS conditional branch instructions:

```
bne $s0, $s1, Lbl     #go to Lbl if $s0!=$s1
beq $s0, $s1, Lbl     #go to Lbl if $s0=$s1
```

Example

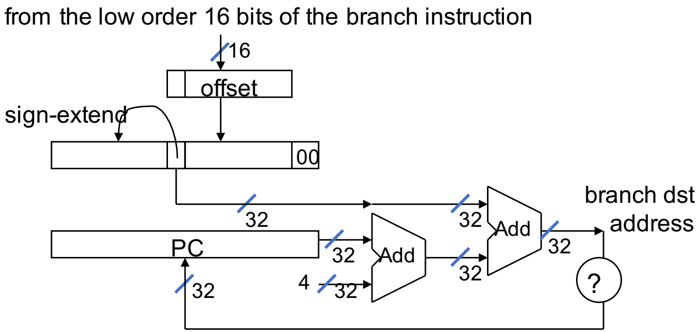
```
       if (i==j) h = i + j;

       bne $s0, $s1, Lbl1
       add $s3, $s0, $s1
Lbl1:  ...
```

- Instruction Format (I format)
- How is the branch destination address specified ?

Specifying Branch Destinations

- Use a register (like in `lw` and `sw`) added to the 16-bit offset
- which register? Instruction Address Register (the PC)
- its use is automatically implied by instruction
- PC gets updated (PC+4) during the fetch cycle so that it holds the address of the next instruction
- limits the branch distance to -2^{15} to $+2^{15} - 1$ (word) instructions from the (instruction after the) branch instruction, but most branches are local anyway



In Support of Branch Instructions

- We have `beq`, `bne`, but what about other kinds of branches (e.g., branch-if-less-than)?
- For this, we need yet another instruction, `slt`

Set on less than instruction:

```
slt $t0, $s0, $s1        # if $s0 < $s1  then
                         # $t0 = 1       else
                         # $t0 = 0
```

- Instruction format (R format)

Alternate versions of `slt`

```
slti  $t0, $s0, 25    # if $s0 < 25 then $t0=1 ...
sltu  $t0, $s0, $s1   # if $s0 < $s1 then $t0=1 ...
sltiu $t0, $s0, 25    # if $s0 < 25 then $t0=1 ...
```

Aside: More Branch Instructions

Can use `slt`, `beq`, `bne`, and the fixed value of 0 in register `$zero` to create other conditions

- less than: `blt  $s1,  $s2,  Label`

```
slt  $at, $s1, $s2          #$at set to 1 if
bne  $at, $zero, Label      #$s1 < $s2
```

- less than or equal to: `ble  $s1,  $s2,  Label`
- greater than: `bgt  $s1,  $s2,  Label`
- great than or equal to: `bge  $s1,  $s2,  Label`
- Such branches are included in the instruction set as pseudo instructions – recognized (and expanded) by the assembler
- It's why the assembler needs a reserved register (`$at`)

Bounds Check Shortcut

- Treating signed numbers as if they were unsigned gives a low cost way of checking if $0 \leq x < y$ (index out of bounds for arrays)

```
sltu $t0, $s1, $t2      # $t0 = 0 if
                        # $s1 > $t2 (max)
                        # or $s1 < 0 (min)
beq  $t0,$zero,IOOB     # go to IOOB if
                        # $t0 = 0
```

- The key is that negative integers in two's complement look like large numbers in unsigned notation.
- Thus, an unsigned comparison of x < y also checks if x is negative as well as if x is less than y.

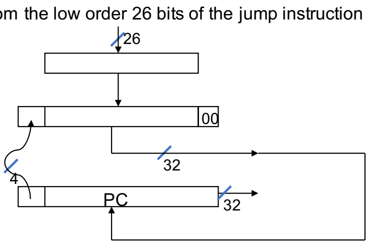
Other Control Flow Instructions

- MIPS also has an unconditional branch instruction or jump instruction:

```
        j   label           #go to label
```

- Instruction Format (J Format)

0x02	26-bit address



EX-2: Branching Far Away

What if the branch destination is further away than can be captured in 16 bits? Re-write the following codes.

```
        beq     $s0, $s1, L1
```

EX: Compiling a while Loop in C

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

Assume that `i` and `k` correspond to registers `$s3` and `$s5` and the base of the array save is in `$s6`.

EX: Compiling a while Loop in C

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

Assume that `i` and `k` correspond to registers `$s3` and `$s5` and the base of the array save is in `$s6`.

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

Note: left shift `$s3` to align word address, and later address is increased by 1

Overview

Six Steps in Execution of a Procedure

1. Main routine (caller) places parameters in a place where the procedure (callee) can access them
   - `$a0` – `$a3`: four argument registers
2. Caller transfers control to the callee
3. Callee acquires the storage resources needed
4. Callee performs the desired task
5. Callee places the result value in a place where the caller can access it
   - `$v0`–`$v1`: two value registers for result values
6. Callee returns control to the caller
   - `$ra`: one return address register to return to the point of origin

Instructions for Accessing Procedures

- MIPS procedure call instruction:

```
    jal    ProcedureAddress    #jump and link
```

- Saves PC+4 in register $ra to have a link to the next instruction for the procedure return
- Machine format (J format):
- Then can do procedure return with a

```
    jr    $ra             #return
```

- Instruction format (R format)

Example of Accessing Procedures

- For a procedure that computes the GCD of two values i (in `$t0`) and j (in `$t1`): `gcd(i,j);`
- The caller puts the i and j (the parameters values) in `$a0` and `$a1` and issues a

```
        jal gcd         #jump to routine gcd
```

- The callee computes the GCD, puts the result in `$v0`, and returns control to the caller using

```
gcd: . . .              #code to compute gcd
         jr   $ra       #return
```

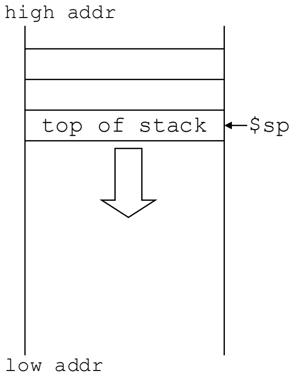
What if the callee needs to use more registers than allocated to argument and return values?

- Use a stack: a last-in-first-out queue
- One of the general registers, `$sp` (`$29`), is used to address the stack
- "grows" from high address to low address
- push: add data onto the stack, data on stack at new `$sp`

```
$sp = $sp - 4
```

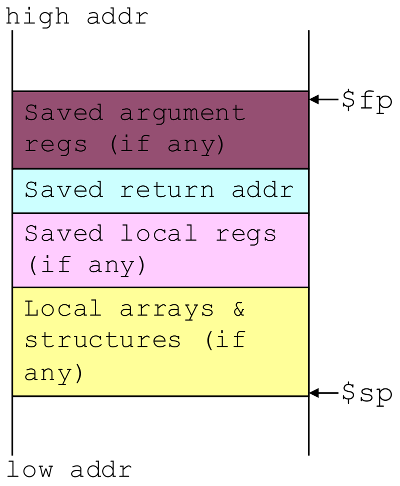
- pop: remove data from the stack, data from stack at `$sp`

```
$sp = $sp + 4
```



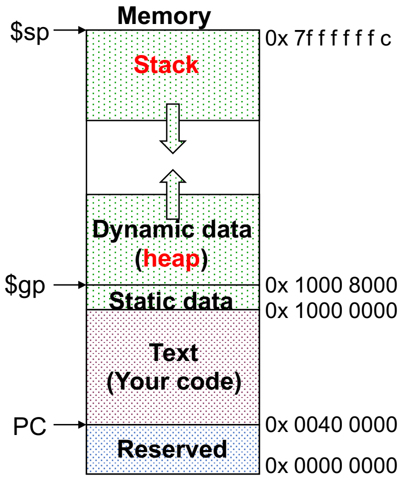
Allocating Space on the Stack

- The segment of the stack containing a procedure's saved registers and local variables is its procedure frame (aka activation record)
- The frame pointer (`$fp`) points to the first word of the frame of a procedure – providing a stable "base" register for the procedure
- `$fp` is initialized using `$sp` on a call and `$sp` is restored using `$fp` on a return


high addr
$fp
Saved argument regs (if any)
Saved return addr
Saved local regs (if any)
Local arrays & structures (if any)
$sp
low addr

Allocating Space on the Heap

- Static data segment for constants and other static variables (e.g., arrays)
- Dynamic data segment (aka heap) for structures that grow and shrink (e.g., linked lists)
- Allocate space on the heap with `malloc()` and free it with `free()` in C



EX-3: Compiling a C Leaf Procedure

Leaf procedures are ones that do not call other procedures. Given the MIPS assembler code for the follows.

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

Solution:

EX-3: Compiling a C Leaf Procedure

Leaf procedures are ones that do not call other procedures. Given the MIPS assembler code for the follows.

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

Solution:

Suppose g, h, i, and j are in `$a0, $a1, $a2, $a3`

```
leaf_ex:  addi   $sp,$sp,-8 #make stack room
          sw     $t1,4($sp) #save $t1 on stack
          sw     $t0,0($sp) #save $t0 on stack
          add    $t0,$a0,$a1
          add    $t1,$a2,$a3
          sub    $v0,$t0,$t1
          lw     $t0,0($sp) #restore $t0
          lw     $t1,4($sp) #restore $t1
          addi   $sp,$sp,8  #adjust stack ptr
          jr     $ra
```

Nested Procedures

- Nested Procedure: call other procedures
- What happens to return addresses with nested procedures?

```
int rt_1 (int i)
{
      if (i == 0) return 0;
      else return rt_2(i-1);
}
```

Nested procedures (cont.)

```
caller: jal  rt_1
next:        . . .

rt_1:   bne  $a0, $zero, to_2
        add  $v0, $zero, $zero
        jr   $ra
to_2:   addi $a0, $a0, -1
        jal  rt_2
        jr   $ra

rt_2:   . . .
```

- On the call to `rt_1`, the return address (next in the caller routine) gets stored in `$ra`.

Question:

What happens to the value in `$ra` (when `$a0!=0`) when `rt_1` makes a call to `rt_2`?

Compiling a Recursive Procedure

A procedure for calculating factorial

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

- A recursive procedure (one that calls itself!)

```
fact (0) = 1
fact (1) = 1 * 1 = 1
fact (2) = 2 * 1 * 1 = 2
fact (3) = 3 * 2 * 1 * 1 = 6
fact (4) = 4 * 3 * 2 * 1 * 1 = 24
. . .
```

- Assume n is passed in `$a0`; result returned in `$v0`

Compiling a Recursive Procedure (cont.)

```
fact: addi   $sp, $sp, -8        #adjust stack pointer
      sw     $ra, 4($sp)         #save return address
      sw     $a0, 0($sp)         #save argument n
      slti   $t0, $a0, 1         #test for n < 1
      beq    $t0, $zero, L1      #if n >=1, go to L1
      addi   $v0, $zero, 1       #else return 1 in $v0
      addi   $sp, $sp, 8         #adjust stack pointer
      jr     $ra                 #return to caller
L1:   addi   $a0, $a0, -1        #n >=1, so decrement n
      jal    fact                #call fact with (n-1)
                                 #this is where fact returns
bk_f: lw     $a0, 0($sp)         #restore argument n
      lw     $ra, 4($sp)         #restore return address
      addi   $sp, $sp, 8         #adjust stack pointer
      mul    $v0, $a0, $v0       #$v0 = n * fact(n-1)
      jr     $ra                 #return to caller
```

Note: `bk_f` is carried out when `fact` is returned.

Question:

Why we don’t load `$ra, $a0` back to registers?

Overview

Atomic Exchange Support

- Need hardware support for synchronization mechanisms to avoid data races where the results of the program can change depending on how events happen to occur
- Two memory accesses from different threads to the same location, and at least one is a write
- Atomic exchange (atomic swap): interchanges a value in a register for a value in memory atomically, i.e., as one operation (instruction)
- Implementing an atomic exchange would require both a memory read and a memory write in a single, uninterruptable instruction.
- An alternative is to have a pair of specially configured instructions

```
ll   $t1, 0($s1)     #load linked
sc   $t0, 0($s1)     #store conditional
```

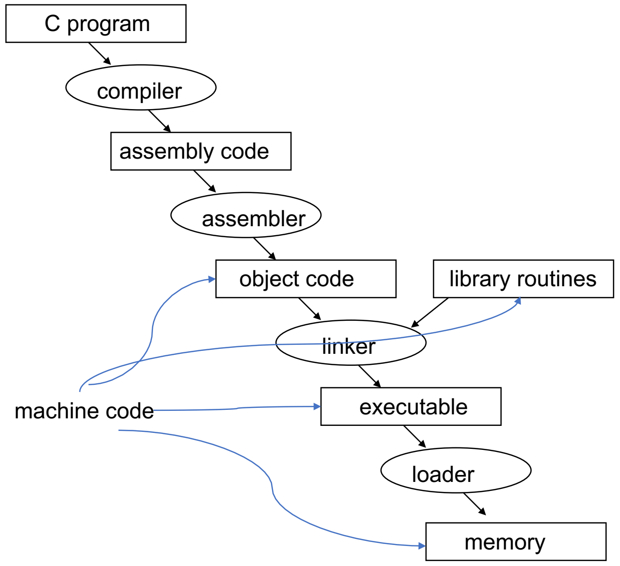
Automic Exchange with `ll` and `sc`

- If the contents of the memory location specified by the `ll` are changed before the `sc` to the same address occurs, the `sc` fails
- If the value in memory between the `ll` and the `sc` instructions changes, then `sc` returns a 0 in `$t0` causing the code sequence to try again.

Example:

```
try:  add $t0, $zero, $s4     #$t0=$s4 (exchange value)
      ll  $t1, 0($s1)         #load memory value to $t1
      sc  $t0, 0($s1)         #try to store exchange
                              #value to memory, if fail
                              #$t0 will be 0
      beq $t0, $zero, try     #try again on failure
      add $s4, $zero, $t1     #load value in $s4
```

The C Code Translation Hierarchy



Compiler Benefits

- Comparing performance for bubble (exchange) sort
- To sort 100,000 words with the array initialized to random values on a Pentium 4 with a 3.06 clock rate, a 533 MHz system bus, with 2 GB of DDR SDRAM, using Linux version 2.4.20

The un-optimized code has the best CPI∗, the O1 version has the lowest instruction count, but the O3 version is the fastest.

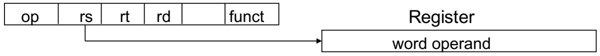
gcc opt	Relative performance	Clock cycles (M)	Instr count (M)	CPI
None	1.00	158,615	114,938	1.38
O1 (medium)	2.37	66,990	37,470	1.79
O2 (full)	2.38	66,521	39,993	1.66
O3 (proc mig)	2.41	65,747	44,993	1.46

∗CPI: clock cycles per instruction

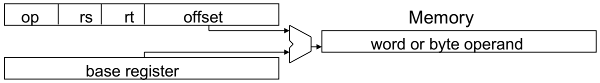
Overview

Addressing Modes Illustrated

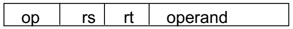
1. Register addressing



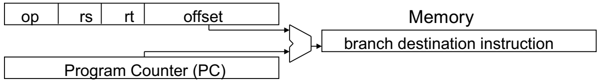
2. Base (displacement) addressing



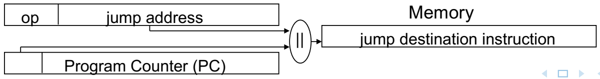
3. Immediate addressing



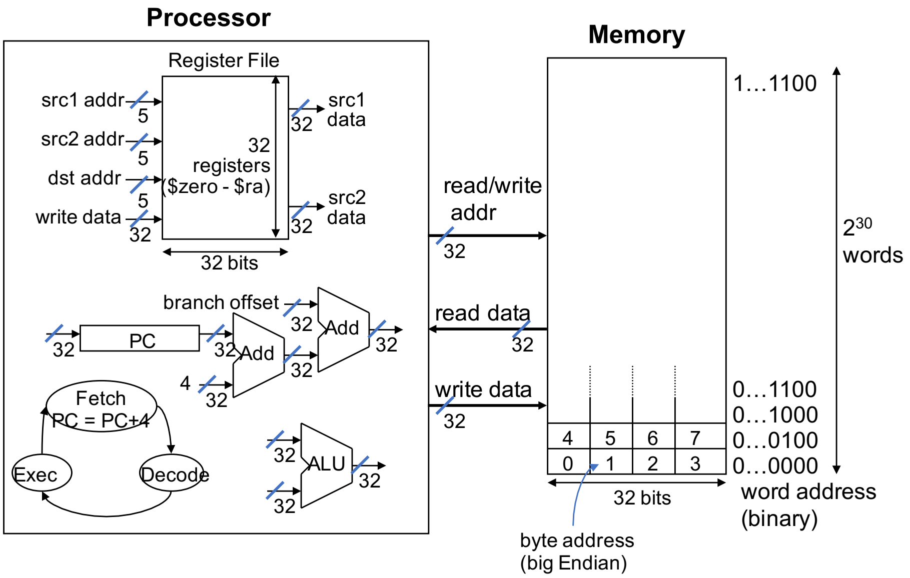
4. PC-relative addressing



5. Pseudo-direct addressing



MIPS Organization So Far



Processor
Register File
src1 addr
5
src2 addr
5
dst addr
5
write data
32
32 registers ($zero - $ra)
src1 data
32
src2 data
32
32 bits
branch offset
32
PC
32
Add
32
4
32
Add
32
Fetch PC = PC+4
Exec
Decode
ALU
32
32
32
Memory
read/write addr
32
read data
32
write data
32
1…1100
0…1100
0…1000
0…0100
0…0000
4 5 6 7
0 1 2 3
32 bits
byte address (big Endian)
2^{30} words
word address (binary)