MIPS Architectural Approach

Load/store or register-register instruction set

- Only operate on data in registers
  - Register operations affect the entire contents of register
    - No partial register writes except for single-precision FP
- Only load/store instructions access memory
- True in all RISC instruction sets
- True in all instruction sets designed since 1980

Emphasis on efficient implementation

- *Make the common case fast*
  - A system can be so simple that it obviously has no bugs, or so complex that it has no obvious bugs. (adapted C. A. R. Hoare)

Simplicity: provide primitives rather than solutions

- *Simplicity favors regularity*

MIPS Data Types

Bit String: sequence of bits of a particular length

- 8 bits is a byte
- 16 bits is a half-word
- 32 bits is a word
- 64 bits is a double-word

Character

- supported as a byte (signed or unsigned)

Integers

- 2's Complement

Floating Point: $M \times 2^E$

- single precision
- double precision

MIPS Storage Model

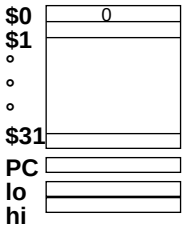
2^{32} bytes of memory: accessible by loads/stores

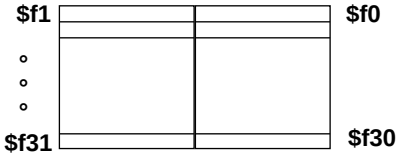
31 x 32-bit GPRs (R0 = 0) or integer multiply/divide

- why only 32 registers? *Smaller is faster*

PC: incremented by 4 for each instruction

- except for branch, j, jal





FP registers are paired for double-precision. Specify the even register, which holds the less-significant word.

MIPS Computational Instructions

Arithmetic/Logical instructions

- Three operand format: result + two sources
- Operands: registers, 16-bit immediates
- Signed & unsigned arithmetic operations:
  - Sign-extension for immediates
  - Trapping of overflow for signed values
- Compare instructions
  - Signed vs. unsigned: comparison is different

Integer multiply/divide

- Use HI/LO registers

Floating Point instructions

- Operate on floating point registers
- Double and single precision
- Typical: add, multiply, divide, subtract

MIPS Integer Arithmetic

add	add $1,$2,$3	$1 = $2 + $3	3 operands; exceptions
subtract	sub $1,$2,$3	$1 = $2 – $3	3 operands; exceptions
add immediate	addi $1,$2,100	$1 = $2 + 100	+ constant; exceptions
add unsigned	addu $1,$2,$3	$1 = $2 + $3	3 operands; no exceptions
subtract unsign	subu $1,$2,$3	$1 = $2 – $3	3 operands; no exceptions
add imm unsign	addiu $1,$2,100	$1 = $2 + 100	+ constant; no exceptions
set less than	slt $1,$2,$3	$1 = ($2 < $3)	compare signed <
set less than imm	slti $1,$2,100	$1 = ($2 < 100)	compare signed < constant
set less than uns	sltu $1,$2,$3	$1 = ($2 < $3)	compare unsigned <
set l. t. imm. uns.	sltiu $1,$2,100	$1 = ($2 < 100)	compare unsigned < const

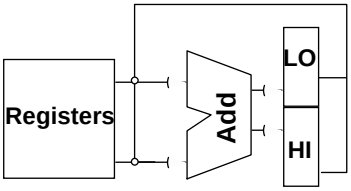
Note: Immediates are sign-extended to form constant for arithmetic operations

MIPS Multiply/Divide

multiply	mult $2,$3	Hi, Lo = $2 x $3	64-bit signed product
multiply unsign	multu $2,$3	Hi, Lo = $2 x $3	64-bit unsigned prod.
divide	div $2,$3	Lo = $2 ÷ $3,	Lo = quotient
		Hi = $2 mod $3	Hi = remainder
divide unsign	divu $2,$3	Lo = $2 ÷ $3,	Unsigned quotient
		Hi = $2 mod $3	Unsigned remainder
Move from Hi	mfhi $1	$1 = Hi	Get copy of Hi
Move from Lo	mflo $1	$1 = Lo	Get copy of Lo

Rationale

- Deal with 64-bit result
- Simplify handling of instruction



MIPS Logical Instructions

Instruction	Example	Meaning	Comment
and	and $1,$2,$3	$1 = $2 & $3	Logical AND
or	or $1,$2,$3	$1 = $2 | $3	Logical OR
xor	xor $1,$2,$3	$1 = $2 ^$3	Logical XOR
nor	nor $1,$2,$3	$1 = ~($2 |$3)	Logical NOR
and immediate	andi $1,$2,10	$1 = $2 & 10	Logical AND w. constant
or immediate	ori $1,$2,10	$1 = $2 | 10	Logical OR w. constant
xor immediate	xori $1, $2,10	$1 = $2 ^10	Logical XOR w. constant
shift left log	sll $1,$2,10	$1 = $2 << 10	Shift left by constant
shift right log	srl $1,$2,10	$1 = $2 >> 10	Shift right by constant
shift right arith	sra $1,$2,10	$1 = $2 >> 10	Shift right (sign extend)
shift left log var	sllv $1,$2,$3	$1 = $2 << $3	Shift left by variable
shift right log var	srlv $1,$2, $3	$1 = $2 >> $3	Shift right by variable
shift right arith	srav $1,$2, $3	$1 = $2 >> $3	Shift right arith. by var
d upper imm	lui $1,40	$1 = 40 << 16	Put imm in upper 16 bits

MIPS Memory Access

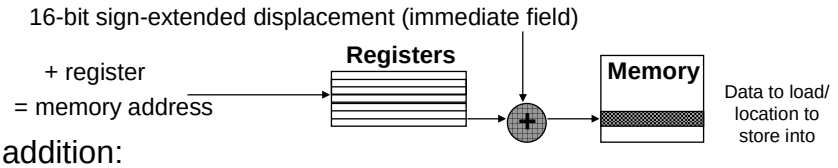
All memory access through loads and stores

Aligned words, halfwords, and bytes

- A halfword or byte loaded from memory can be sign- or zero-extended to form a word in the destination register

Floating-point loads/stores for FP registers

Single addressing mode (displacement or based)



In addition:

- Displacement = 0 uses register contents as address
- Register = 0 uses 16-bit displacement as address

MIPS Load/Store Instructions

Instruction	Example	Meaning	Comments
store word	sw $3, 8($4)	Mem[$4+8]=$3	Store word
store halfword	sh $3, 6($2)	Mem[$2+6]=$3	Stores only lower 16 bits
store byte	sb $2, 7($3)	Mem[$3+7]=$3	Stores only lowest byte
store float	sf $f2, 4($2)	Mem[$2+4]=$f2	Store FP word
load word	lw $1, 8($2)	$1=Mem[8+$2]	Load word
load halfword	lh $1, 6($3)	$1=Mem[6+$3]	Load half; sign extend
load half unsign	lhu $1, 6($3)	$1=Mem[6+$3]	Load half; zero extend
load byte	lb $1, 5($3)	$1=Mem[5+$3]	Load byte; sign extend
load byte unsign	lbu $1, 5($3)	$1=Mem[5+$3]	Load byte; zero extend
load float	lf F1, 4($3)	$f1=Mem[4+$3]	Load FP register

Forming a Memory Address

Let's say you want to load a value from a fixed location in memory, known at compile time

Address: 0x123450

```
lui  $1, 0x12          # $1 = upper 16 bits of constant
addiu $1, $1, 0x3450   # add in lower 16 bits
lw   $2, 0($1)         # perform the load
```

Not taking advantage of displacement capability

```
lui  $1, 0x12          # $1 = upper 16 bits of constant
lw   $2, 0x3450($1)    # perform the load
```

MIPS Branch/Jump Instructions

Two classes:

- Jumps
  - Unconditional, not PC-relative
  - For procedure call, unconditional control, switch statements, simulating long branches
- Branches
  - Conditional and PC relative
  - For conditional control and PC-relative unconditional

Jumps

Instruction	Example	Meaning	Comment
jump	j 10000	PC = 40000	jump to address
jump register	jr $31	PC = $31	jump to addr in register
jump and link	jal 10000	$31 = PC + 4;	Save PC next instruction
		PC = 40000	jump to address

MIPS Branches

Conditional branch is compare-and-branch

- Conditions:
  - Comparison against 0: equality, sign-test
  - Comparison of two registers: equality only
  - Remaining set of compare-and-branch take two instructions

Instruction	Example	Meaning
branch equal	beq $1,$2,100	if ($1 == $2) PC=PC+4+400
branch not eq	bne $1,$2,100	if ($1 != $2) PC=PC+4+400
---	---	---
branch l.t. 0	bltz $1,100	if ($1 < 0) PC = PC+4+400
branch g.t./eq 0	bgez $1,100	if ($1 >= 0) PC = PC+4+400

Programming Example: Searching

C source code:

```
count=0;
for (index=head; index<=n; index++)
    if (C[index] == target) count ++;
```

MIPS assembly code, assuming:

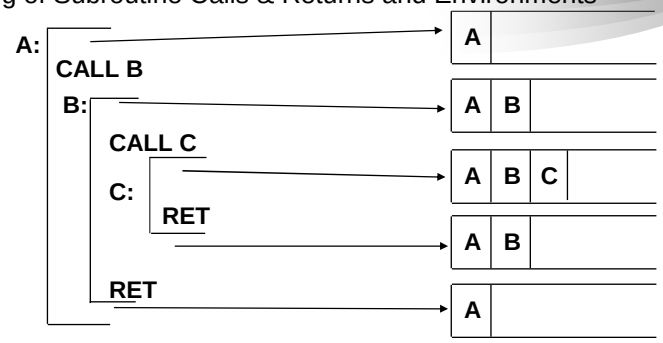
- count in $5, index in $6, head in $1, addr of C in $2, target in $3; n in $4

```
      li    $5,0          # set count =0 (addiu $5,$0,0)
      move  $6,$1         # initial index (addu $6,$1,$0)
loop: slt   $9,$4,$6      # $9=1 if n < index (index > n)
      bne   $9,$0,exit    # if index>n goto exit label
      sll   $7,$6,2       # multiply index by 4
      addu  $7,$7,$2      # address of C [index]
      lw    $8,0($7)      # C[index] = $8
      bne   $8,$3,next    # test if equal
      addiu $5,$5,1       # increment count
next: addiu $6,$6,1       # increment index
      j     loop          # unconditional jump to loop
exit:
```

- Straightforward, but not best (smallest or fastest) code!

Stacks

Stacking of Subroutine Calls & Returns and Environments



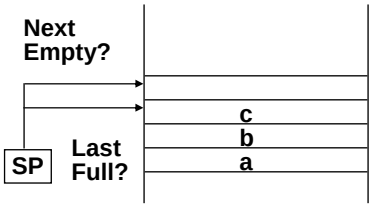
Stacks are a natural structure for procedure calls / local variables

Implementing the stack

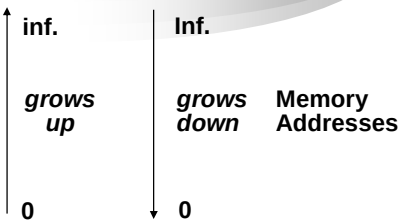
- Common rules are needed across procedures
- Recent machines use software convention
- Some earlier machines use hardware mechanisms and instructions

Stack Frames

How is empty stack represented?



Stacks can grow up or down:



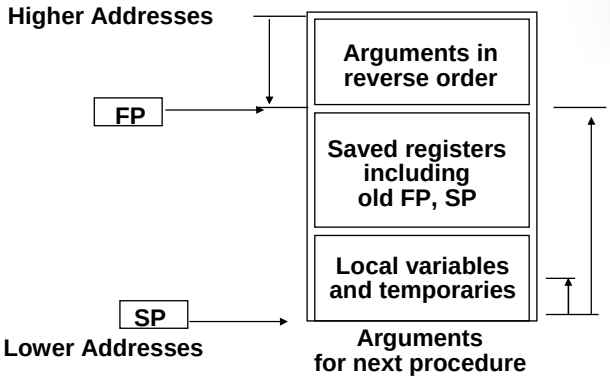
Down/Next Empty (MIPS)

POP: Increment (SP) **PUSH: Write to Mem(SP)**

Read from Mem(SP) **Decrement SP**

Single elements are not usually pushed/popped. Instead an entire stack frame may be pushed or popped in one increment/decrement.

Stack Frame Layout



Access arguments at fixed offset from FP. Allow variable number of arguments.

Access local variables and saved registers at fixed (positive) offset from SP

Parameters are passed in registers; extras on stack

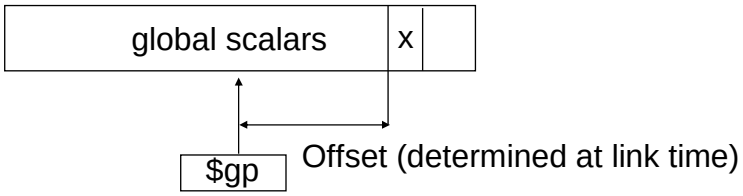
Compilers try to keep scalar variables in registers, not memory

- Stack locations for spilling/saving on procedure calls

MIPS Addressing Model

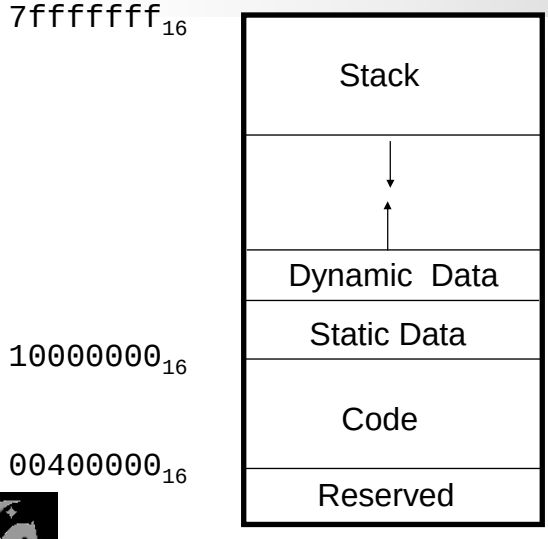
Local scalar variables of a procedure stored in registers
- Spilled to stack frame
- Space allocated when compiled
- Loaded/stored into registers as needed

Global static scalar variables (single variables, not arrays)
- Allocated in a 64KB static area at compile time
- Addressed with a register pointing into area + offset



Dynamic allocated in heap, reserved memory below stack

MIPS Address Map



Stack and dynamic area grow towards one another to maximize storage use before collision.

MIPS Software Register Convention

0	$0	zero constant 0
1	$at	reserved for assembler
2	$v0	expression evaluation &
3	$v1	function results
4	$a0	arguments (caller saves)
5	$a1	
6	$a2	
7	$a3	
8	$t0	temporary: caller saves
. . .		
15	$t7	
16	$s0	callee saves
. . .		
23	$s7	
24	$t8	temporary (cont'd)
25	$t9	
26	$k0	reserved for OS kernel
27	$k1	
28	$gp	global pointer
29	$sp	stack pointer
30	$fp	frame pointer
31	$ra	Return Address (HW)

MIPS Register Saving Convention

Preserved on Call	Not Preserved on Call
Saved Registers ($s0..$s7)	Argument Registers ($a0..$a3)
Stack Pointer ($sp)	Return Value Regs ($v0..$v1)
Frame Pointer ($fp)	Temporaries ($t0..$t9)
Return Address ($ra)	
Global Pointer ($gp)	

Preserved registers must be saved and restored by called procedure if modified
Unpreserved registers must be saved by caller if needed after call completes

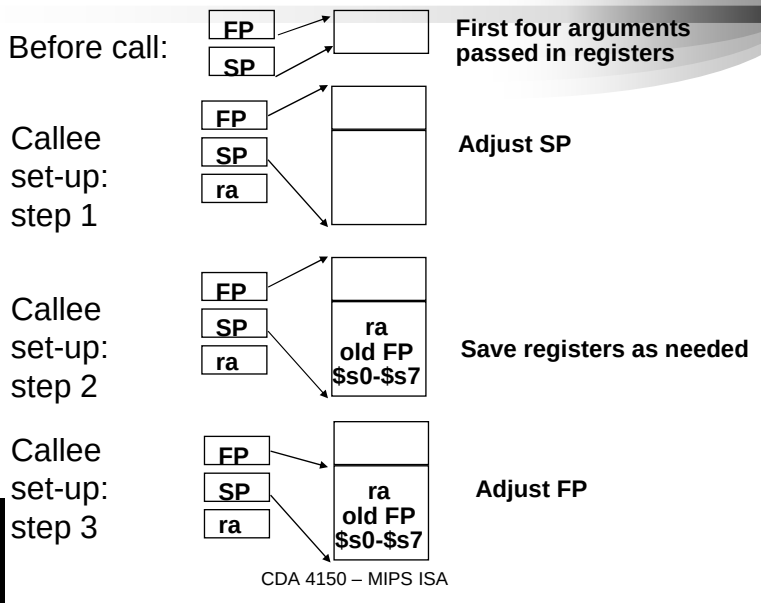
MIPS Calling Convention

Caller
- Save caller-saved registers: $a0–$a3, $v0-$v1 $t0-$t9 if used
- Load arguments: first four in $a0–$a3, rest on stack
- Execute jal instruction

Callee
- Allocate memory in frame: $sp = $sp – frame size
- Save callee-saved registers $s0–$s7,$fp,$ra if used
- Create frame: $fp = $sp + frame size - 4

Return
- Place return value in $v0
- Restore any callee-saved registers
- Pop stack: $sp = $sp+frame size
- Return by jr $ra

MIPS Calling Convention Steps



MIPS Calling Convention Example

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

```
fact: slti $t0,$a0,2   ; test n < 2
   beq  $t0,$0,skip
   li   $v0,1          ; return value
   jr   $ra            ; return
skip:  addiu  $sp,$sp,-32 ; create frame
   sw   $ra,20($sp)    ; save $ra
   sw   $fp,16($sp)    ; save $fp
   addiu $fp,$sp,28    ; set $fp
   sw   $a0,0($fp)     ; save n
   addiu       $a0,$a0,-1     ; n-1
   jal  fact
   lw   $a0,0($fp)     ; restore n
   mult $v0,$v0,$a0    ; n*fact(n-1)
   lw   $ra,20($sp)    ; restore $ra
   lw   $fp,16($sp)    ; restore $fp
   addiu $sp,$sp,32    ; pop stack
   jr   $ra            ; return
```

MIPS Instruction Encoding (I)

3 formats, all 32 bits in length

Fixed 6-bit opcode begins each instruction

ALU Format (also R format): one opcode

- Register-register ALU instructions

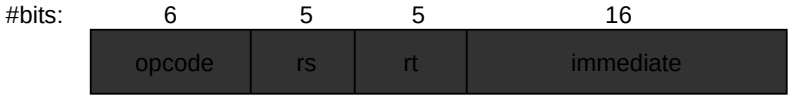
Function code

- Detailed opcode: add, sub, or, and, …

MIPS Instruction Encoding (II)

Immediate instruction format (I format)

- Loads/stores (incl. floating point) sign-extend imm
- Immediate instructions (e.g. addi, lui, etc.)
  - Sign-extend immediate for arithmetic ops (even addu)
  - Zero-extend for logical ops
- Branches sign-extend immediate and scale by 4
  - Add displacement to PC+4
- Different opcode for each instruction



First source or base register

Second source or result register

MIPS Instruction Encoding (III)

Jump format (J format)

- Used for j, jal
- 26-bit offset is scaled by 4 to form 28 lsbs of new PC
  - 4 msbs of new PC copied from current PC

"pseudo-direct" jump target address

MIPS ISA Details (I)

Register 0 is *always* 0 (even if you try to write it)

Jump and link (jal) puts the return address (PC+8) into the link register (R31)

All insts change all 32 bits of the dest register

- Including lui, lb, lh

All read all 32 bits of sources (and, sub, and, or, …)

Data from sub-word loads extended as follows

- lbu, lhu, zero-extended
- lb, lh, sign-extended

MIPS ISA Details (II)

The MIPS architecture defines a

- Branch delay slot
  - Instruction after branch is always executed
- Load delay slot
  - Value returned from load cannot be used the next cycle

The reason for restrictions will be clear next week

Makes perfect sense for simple in-order pipelined machines

Architecture definition

- Every implementation must obey these rules
- Branch delay slot is a burden for the R10000+
- Load delay slot is unnecessary

MIPS Summary

Reduced Instruction Set Computing (RISC) vs. Complex Instruction Set Computing (CISC)

- Terms coined by Patterson and Ditzel (1980)
- Widely-used terms, poorly defined
  - "A RISC processor is any with an instruction set defined after 1980"

Common attributes

- Fixed-length instructions
  - Some embedded processors use variable length instructions to reduce cost
- Load/store architecture (for memory accesses)
- "Large" general-purpose register file (>=32)
- "Simple" operations that can be directly controlled
- One register is hardwired to 0