CS152
Computer Architecture and Engineering
Lecture 2

August 29, 1997

Dave Patterson (http.cs.berkeley.edu/~patterson)

lecture slides: http://www-inst.eecs.berkeley.edu/~cs152/

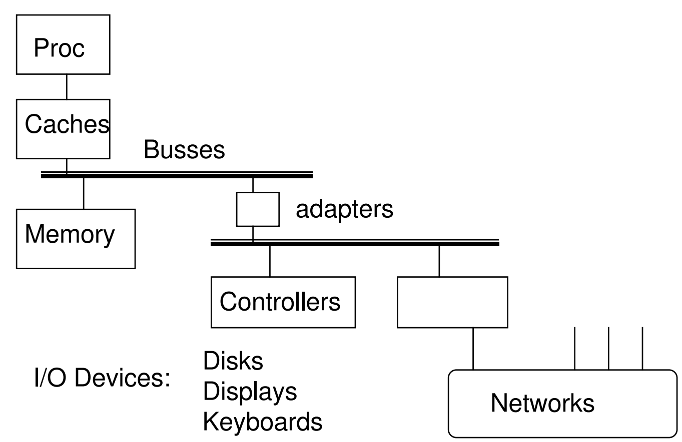
Overview of Today's Lecture: Review Instruction Sets, Performance

- Review from Last Lecture (1 minute)
- Classes, Addressing, Format (20 minutes)
- Administrative Matters (3 minutes)
- Operations, Branching, Calling conventions (25 minutes)
- Break (5 minutes)
- MIPS Details, Performance (25 minutes)

Review: Organization

- All computers consist of five components
  - Processor: (1) datapath and (2) control
  - (3) Memory
  - (4) Input devices and (5) Output devices
- Not all "memory" are created equally
  - Cache: fast (expensive) memory are placed closer to the processor
  - Main memory: less expensive memory--we can have more
- Input and output (I/O) devices have the messiest organization
  - Wide range of speed: graphics vs. keyboard
  - Wide range of requirements: speed, standard, cost ...
  - Least amount of research (so far)

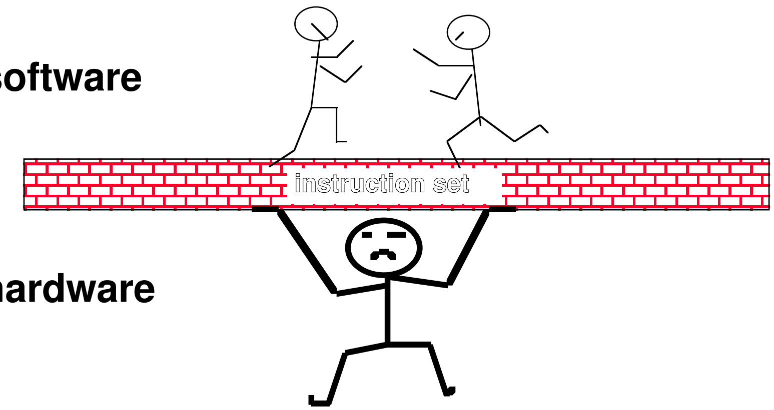
Summary: Computer System Components



- All have interfaces & organizations

Review: Instruction Set Design

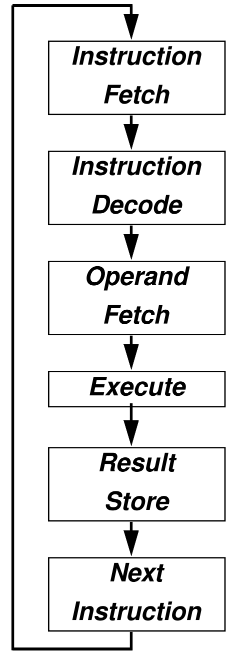
software



hardware

Which is easier to change?

Instruction Set Architecture: What Must be Specified?



- Instruction Format or Encoding
  - how is it decoded?
- Location of operands and result
  - where other than memory?
  - how many explicit operands?
  - how are memory operands located?
  - which can or cannot be in memory?
- Data type and Size
- Operations
  - what are supported
- Successor instruction
  - jumps, conditions, branches
  - *fetch-decode-execute is implicit!*

Basic ISA Classes

Accumulator (1 register):

1 address	add A	acc ← acc + mem[A]
1+x address	addx A	acc ← acc + mem[A + x]

Stack:

0 address	add	tos ← tos + next

General Purpose Register:

2 address	add A B	EA(A) ← EA(A) + EA(B)
3 address	add A B C	EA(A) ← EA(B) + EA(C)

Load/Store:

3 address	add Ra Rb Rc	Ra ← Rb + Rc
	load Ra Rb	Ra ← mem[Rb]
	store Ra Rb	mem[Rb] ← Ra

Comparison:

Bytes per instruction? Number of Instructions? Cycles per instruction?

Comparing Number of Instructions

- Code sequence for C = A + B for four classes of instruction sets:

Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A	Load A	Load R1,A	Load R1,A
Push B	Add B	Add R1,B	Load R2,B
Add	Store C	Store C, R1	Add R3,R1,R2
Pop C			Store C,R3

General Purpose Registers Dominate

- 1975-1995 all machines use general purpose registers
- Advantages of registers
  - registers are faster than memory
  - registers are easier for a compiler to use
    - e.g., (A*B) – (C*D) – (E*F) can do multiplies in any order vs. stack
  - registers can hold variables
    - memory traffic is reduced, so program is sped up (since registers are faster than memory)
    - code density improves (since register named with fewer bits than memory location)

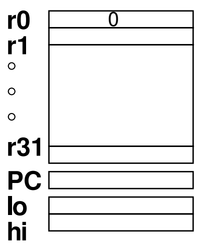
Summary on Instruction Classes

- Expect new instruction set architecture to use
- general purpose register

Pipelining => Expect it to use load store variant of GPR ISA

MIPS I Registers

- Programmable storage
  - 2^32 x bytes of memory
  - 31 x 32-bit GPRs (R0 = 0)
  - 32 x 32-bit FP regs (paired DP)
  - HI, LO, PC



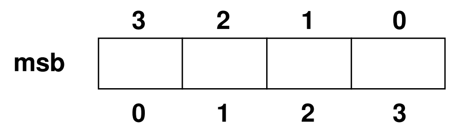
Memory Addressing

- Since 1980 almost every machine uses addresses to level of 8-bits (byte)
- 2 questions for design of ISA:
  - Since could read a 32-bit word as four loads of bytes from sequential byte addresses or as one load word from a single byte address,
    how do byte addresses map onto words?
  - Can a word be placed on any byte boundary?

Addressing Objects: Endianess and Alignment

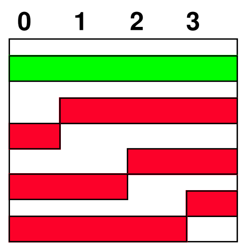
- Big Endian: address of most significant byte = word address (xx00 = Big End of word)
  - IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- Little Endian:address of least significant byte = word address (xx00 = Little End of word)
  - Intel 80x86, DEC Vax, DEC Alpha (Windows NT)

little endian byte 0



big endian byte 0

Alignment: require that objects fall on address that is multiple of their size.



Addressing Modes

Addressing mode	Example	Meaning
Register	Add R4,R3	R4← R4+R3
Immediate	Add R4,#3	R4 ← R4+3
Displacement	Add R4,100(R1)	R4 ← R4+Mem[100+R1]
Register indirect	Add R4,(R1)	R4 ← R4+Mem[R1]
Indexed / Base	Add R3,(R1+R2)	R3 ← R3+Mem[R1+R2]
Direct or absolute	Add R1,(1001)	R1 ← R1+Mem[1001]
Memory indirect	Add R1,@(R3)	R1 ← R1+Mem[Mem[R3]]
Auto-increment	Add R1,(R2)+	R1 ← R1+Mem[R2]; R2 ← R2+d
Auto-decrement	Add R1,–(R2)	R2 ← R2–d; R1 ← R1+Mem[R2]
Scaled	Add R1,100(R2)[R3]	R1 ← R1+Mem[100+R2+R3*d]

Why Auto-increment/decrement? Scaled?

Addressing Mode Usage? (ignore register mode)

3 programs measured on machine with all address modes (VAX)

--- Displacement: 42% avg, 32% to 55% ↕ 75% ↕ 85%

--- Immediate: 33% avg, 17% to 43%

--- Register deferred (indirect): 13% avg, 3% to 24%

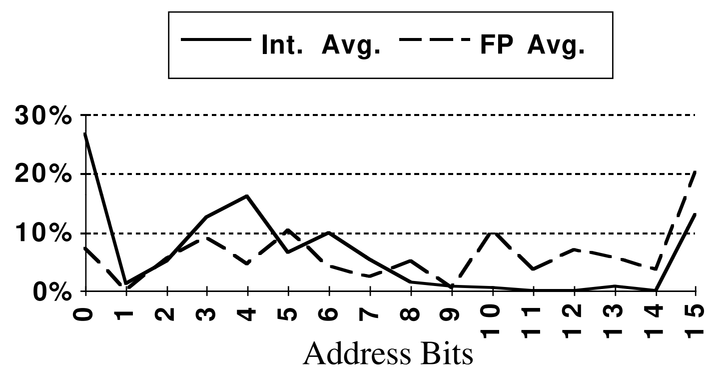
--- Scaled: 7% avg, 0% to 16%

--- Memory indirect: 3% avg, 1% to 6%

--- Misc: 2% avg, 0% to 3%

75% displacement & immediate
88% displacement, immediate & register indirect

Displacement Address Size?



- Avg. of 5 SPECint92 programs v. avg. 5 SPECfp92 programs
- X-axis is in powers of 2: 4 => addresses > 2^3 (8) and ≤ 2^4 (16)
- 1% of addresses > 16-bits
- 12 - 16 bits of displacement needed

Immediate Size?

- 50% to 60% fit within 8 bits
- 75% to 80% fit within 16 bits

Addressing Summary

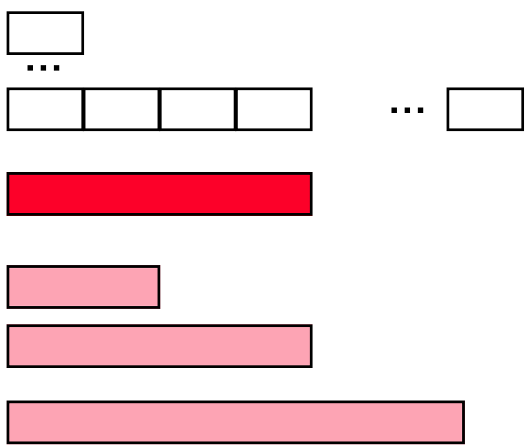
- Data Addressing modes that are important:
  Displacement, Immediate, Register Indirect
- Displacement size should be 12 to 16 bits
- Immediate size should be 8 to 16 bits

Generic Examples of Instruction Format Widths

Variable:

Fixed:

Hybrid:



Summary of Instruction Formats

- If code size is most important,
  use variable length instructions
- If performance is over is most important,
  use fixed length instructions
- Recent embedded machines (ARM, MIPS) added optional mode to execute subset of 16-bit wide instructions (Thumb, MIPS16); per procedure decide performance or density

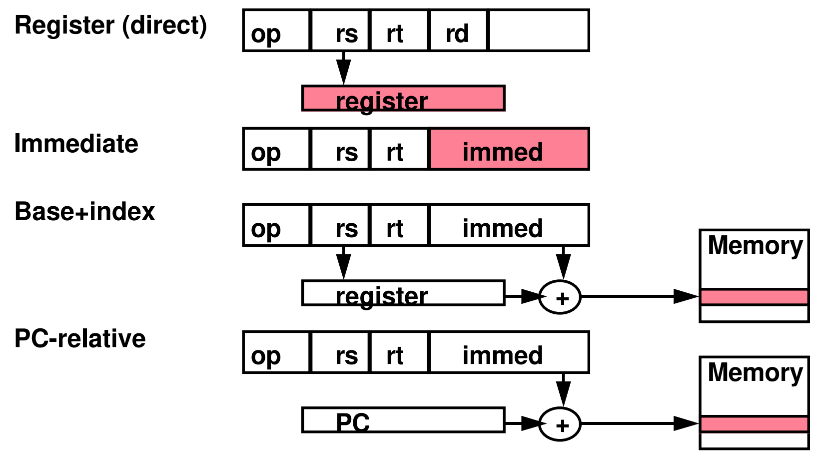
Instruction Format

- If have many memory operands per instructions and many addressing modes,
=>Address Specifier per operand

- If have load-store machine with 1 address per instr. and one or two addressing modes,
=> encode addressing mode in the opcode

MIPS Addressing Modes/Instruction Formats

- All instructions 32 bits wide



- Register Indirect?

Administrative Matters

CS152 news group: ucb.class.cs152
(email patterson@cs with specific questions)

- Slides, handouts available via WWW:
http://www-inst.eecs.berkeley.edu/~cs152/fa97

◦ Video tapes of lectures available for viewing in 205 McLaughlin

- Prerequisite quiz Friday September 5: CS 61C, CS 150
- Review Chapters 1-4 of COD:HSI Second Edition
- Turn in survey forms with photo

Typical Operations (little change since 1960)

Data Movement	Load (from memory) Store (to memory) memory-to-memory move register-to-register move input (from I/O device) output (to I/O device) push, pop (to/from stack)
Arithmetic	integer (binary + decimal) or FP Add, Subtract, Multiply, Divide
Shift	shift left/right, rotate left/right
Logical	not, and, or, set, clear
Control (Jump/Branch)	unconditional, conditional
Subroutine Linkage	call, return
Interrupt	trap, return
Synchronization	test & set (atomic r-m-w)
String	search, translate
Graphics (MMX)	parallel subword ops (4 16bit add)

Top 10 80x86 Instructions

Rank	instruction	Integer Average Percent total executed
1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move register-register	4%
9	call	1%
10	return	1%
	Total	96%

- Simple instructions dominate instruction frequency

Operation Summary

- Support these simple instructions, since they will dominate the number of instructions executed:

load,
store,
add,
subtract,
move register-register,
and,
shift,
compare equal, compare not equal,
branch,
jump,
call,
return;

Compilers and Instruction Set Architectures

- Ease of compilation
  - orthogonality: no special registers, few special cases, all operand modes available with any data type or instruction type
  - completeness: support for a wide range of operations and target applications
  - regularity: no overloading for the meanings of instruction fields
  - streamlined: resource needs easily determined
- Register Assignment is critical too
  - Easier if lots of registers

Summary of Compiler Considerations

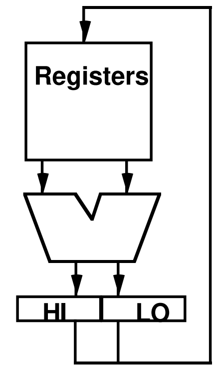
- Provide at least 16 general purpose registers plus separate floating-point registers,
- Be sure all addressing modes apply to all data transfer instructions,
- Aim for a minimalist instruction set.

MIPS I Operation Overview

- Arithmetic logical
- Add, AddU, Sub, SubU, And, Or, Xor, Nor, SLT, SLTU
- AddI, AddIU, SLTI, SLTIU, AndI, OrI, XorI, LUI
- SLL, SRL, SRA, SLLV, SRLV, SRAV
- Memory Access
- LB, LBU, LH, LHU, LW, LWL,LWR
- SB, SH, SW, SWL, SWR

Multiply / Divide

- Start multiply, divide
  - MULT rs, rt
  - MULTU rs, rt
  - DIV rs, rt
  - DIVU rs, rt
- Move result from multiply, divide
  - MFHI rd
  - MFLO rd
- Move to HI or LO
  - MTHI rd
  - MTLO rd
- Why not Third field for destination?
  (Hint: how many clock cycles for multiply or divide vs. add?)



Data Types

Bit: 0, 1

Bit String: sequence of bits of a particular length
4 bits is a nibble
8 bits is a byte
16 bits is a half-word
32 bits is a word
64 bits is a double-word

Character:
ASCII 7 bit code

Decimal:
digits 0-9 encoded as 0000b thru 1001b
two decimal digits packed per 8 bit byte

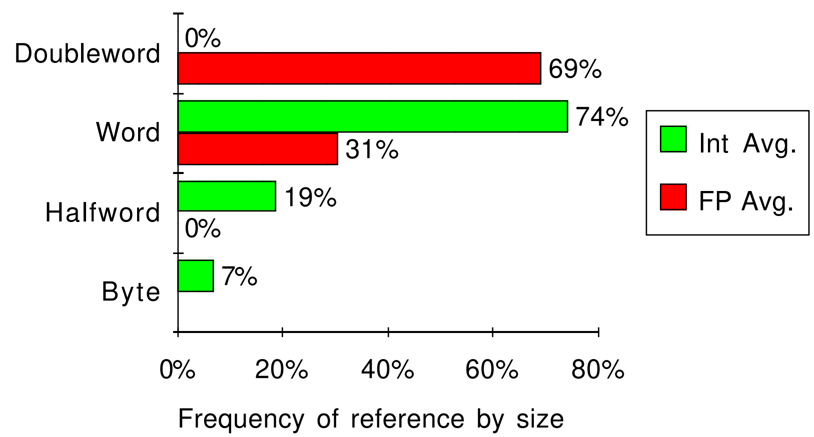
Integers:
2's Complement

Floating Point:
Single Precision
Double Precision
Extended Precision

$M \times R^E$ — M: mantissa, R: base, E: exponent

How many +/- #'s?
Where is decimal pt?
How are +/- exponents represented?

Operand Size Usage



- Support these data sizes and types:
  8-bit, 16-bit, 32-bit integers and
  32-bit and 64-bit IEEE 754 floating point numbers

MIPS arithmetic instructions

Instruction	*Example*	*Meaning*	*Comments*
add	add $1,$2,$3	$1 = $2 + $3	3 operands; exception possible
subtract	sub $1,$2,$3	$1 = $2 – $3	3 operands; exception possible
add immediate	addi $1,$2,100	$1 = $2 + 100	+ constant; exception possible
add unsigned	addu $1,$2,$3	$1 = $2 + $3	3 operands; no exceptions
subtract unsigned	subu $1,$2,$3	$1 = $2 – $3	3 operands; no exceptions
add imm. unsign.	addiu $1,$2,100	$1 = $2 + 100	+ constant; no exceptions
multiply	mult $2,$3	Hi, Lo = $2 x $3	64-bit signed product
multiply unsigned	multu$2,$3	Hi, Lo = $2 x $3	64-bit unsigned product
divide	div $2,$3	Lo = $2 ÷ $3, Hi = $2 mod $3	Lo = quotient, Hi = remainder
divide unsigned	divu $2,$3	Lo = $2 ÷ $3, Hi = $2 mod $3	Unsigned quotient & remainder
Move from Hi	mfhi $1	$1 = Hi	Used to get copy of Hi
Move from Lo	mflo $1	$1 = Lo	Used to get copy of Lo

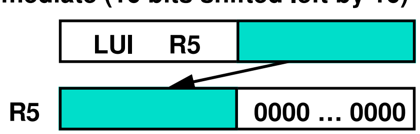
Which add for address arithmetic? Which add for integers?

MIPS logical instructions

Instruction	***Example***	***Meaning***	***Comment***
and	and $1,$2,$3	$1 = $2 & $3	3 reg. operands; Logical AND
or	or $1,$2,$3	$1 = $2 | $3	3 reg. operands; Logical OR
xor	xor $1,$2,$3	$1 = $2 ⊕ $3	3 reg. operands; Logical XOR
nor	nor $1,$2,$3	$1 = ~($2 |$3)	3 reg. operands; Logical NOR
and immediate	andi $1,$2,10	$1 = $2 & 10	Logical AND reg, constant
or immediate	ori $1,$2,10	$1 = $2 | 10	Logical OR reg, constant
xor immediate	xori $1, $2,10	$1 = ~$2 &~10	Logical XOR reg, constant
shift left logical	sll $1,$2,10	$1 = $2 << 10	Shift left by constant
shift right logical	srl $1,$2,10	$1 = $2 >> 10	Shift right by constant
shift right arithm.	sra $1,$2,10	$1 = $2 >> 10	Shift right (sign extend)
shift left logical	sllv $1,$2,$3	$1 = $2 << $3	Shift left by variable
shift right logical	srlv $1,$2, $3	$1 = $2 >> $3	Shift right by variable
shift right arithm.	srav $1,$2, $3	$1 = $2 >> $3	Shift right arith. by variable

MIPS data transfer instructions

Instruction	***Comment***
SW 500(R4), R3	Store word
SH 502(R2), R3	Store half
SB 41(R3), R2	Store byte
LW R1, 30(R2)	Load word
LH R1, 40(R3)	Load halfword
LHU R1, 40(R3)	Load halfword unsigned
LB R1, 40(R3)	Load byte
LBU R1, 40(R3)	Load byte unsigned
LUI R1, 40	Load Upper Immediate (16 bits shifted left by 16)



Why need LUI?

Methods of Testing Condition

° **Condition Codes**

Processor status bits are set as a side-effect of arithmetic instructions (possibly on Moves) or explicitly by compare or test instructions.

ex: add r1, r2, r3
bz label

° **Condition Register**

Ex: cmp r1, r2, r3
bgt r1, label

° **Compare and Branch**

Ex: bgt r1, r2, label

Condition Codes

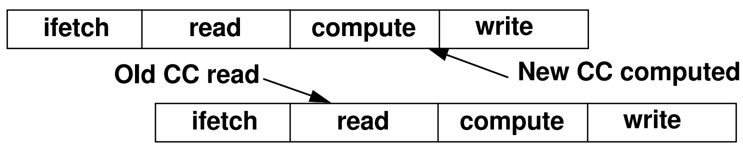
Setting CC as side effect can reduce the # of instructions

```
X:   .                      X:   .
     .           vs.             .
     .                           .
  SUB  r0, #1, r0              SUB  r0, #1, r0
  BRP  X                       CMP  r0, #0
                               BRP  X
```

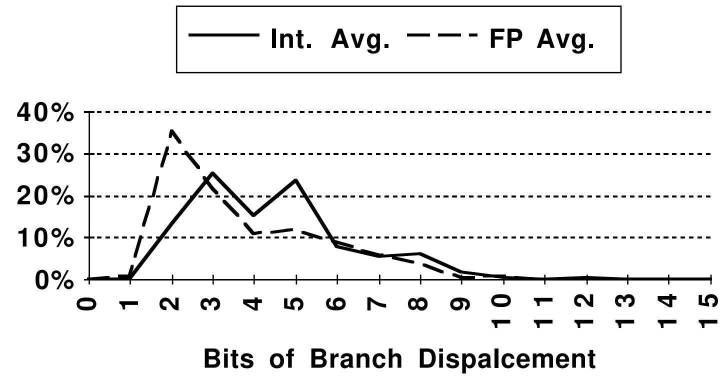
But also has disadvantages:

--- not all instructions set the condition codes; which do and which do not often confusing! *e.g., shift instruction sets the carry bit*

--- dependency between the instruction that sets the CC and the one that tests it: to overlap their execution, may need to separate them with an instruction that does not change the CC



Conditional Branch Distance



• Distance from branch in instructions 2i => $\le \pm 2^{i-1}$ & > 2^{i-2}

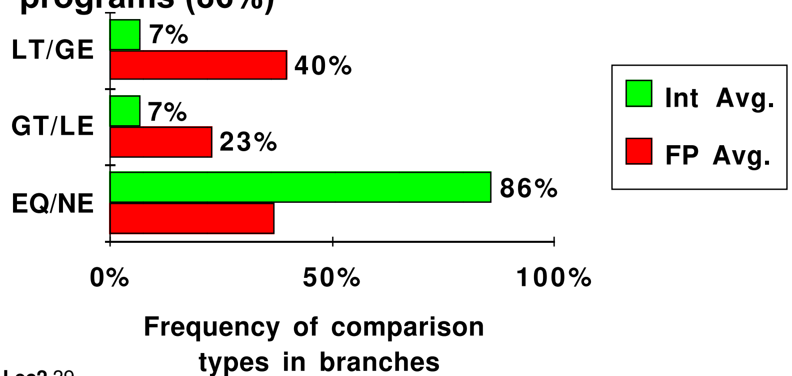
• 25% of integer branches are > 2 to ≤ 4 or -2 to -4 instructions

Conditional Branch Addressing

• PC-relative since most branches are relatively close to the current PC address

• At least 8 bits suggested (± 128 instructions)

• Compare Equal/Not Equal most important for integer programs (86%)



MIPS Compare and Branch

° Compare and Branch
  - BEQ rs, rt, offset   if R[rs] == R[rt] then PC-relative branch
  - BNE rs, rt, offset   <>

° Compare to zero and Branch
  - BLEZ rs, offset   if R[rs] <= 0 then PC-relative branch
  - BGTZ rs, offset   >
  - BLT   <
  - BGEZ   >=
  - BLTZAL rs, offset   if R[rs] < 0 then branch and link (into R 31)
  - BGEZAL   >=

° Remaining set of compare and branch take two instructions

° Almost all comparisons are against zero!

MIPS jump, branch, compare instructions

Instruction	*Example*	*Meaning*
branch on equal	beq \$1,\$2,100	if (\$1 == \$2) go to PC+4+100 *Equal test; PC relative branch*
branch on not eq.	bne \$1,\$2,100	if (\$1!= \$2) go to PC+4+100 *Not equal test; PC relative*
set on less than	slt \$1,\$2,\$3	if (\$2 < \$3) \$1=1; else \$1=0 *Compare less than; 2's comp.*
set less than imm.	slti \$1,\$2,100	if (\$2 < 100) \$1=1; else \$1=0 *Compare < constant; 2's comp.*
set less than uns.	sltu \$1,\$2,\$3	if (\$2 < \$3) \$1=1; else \$1=0 *Compare less than; natural numbers*
set l. t. imm. uns.	sltiu \$1,\$2,100	if (\$2 < 100) \$1=1; else \$1=0 *Compare < constant; natural numbers*
jump	j 10000	go to 10000 *Jump to target address*
jump register	jr \$31	go to \$31 *For switch, procedure return*
jump and link	jal 10000	\$31 = PC + 4; go to 10000 *For procedure call*

Signed vs. Unsigned Comparison

Value?
2's comp   Unsigned?

R1= 0…00 0000 0000 0000 0001 two

R2= 0…00 0000 0000 0000 0010 two

R3= 1…11 1111 1111 1111 1111 two

- **After executing these instructions:**

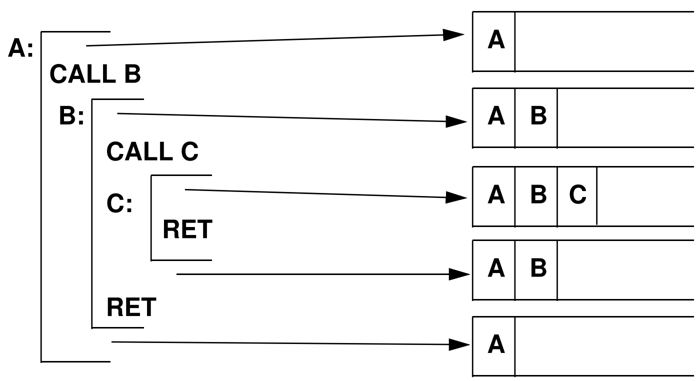
```
slt  r4,r2,r1 ; if (r2 < r1) r4=1; else r4=0
slt  r5,r3,r1 ; if (r3 < r1) r5=1; else r5=0
sltu r6,r2,r1 ; if (r2 < r1) r6=1; else r6=0
sltu r7,r3,r1 ; if (r3 < r1) r7=1; else r7=0
```

- **What are values of registers r4 - r7? Why?**

r4 = ; r5 = ; r6 = ; r7 = ;

Calls: Why Are Stacks So Great?

Stacking of Subroutine Calls & Returns and Environments:



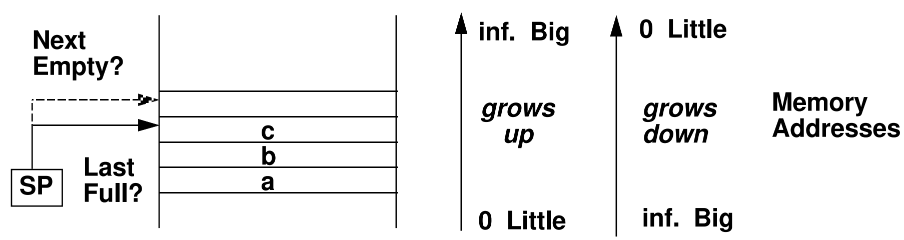
Some machines provide a memory stack as part of the architecture (e.g., VAX)

Sometimes stacks are implemented via software convention (e.g., MIPS)

Memory Stacks

Useful for stacked environments/subroutine call & return even if operand stack not part of architecture

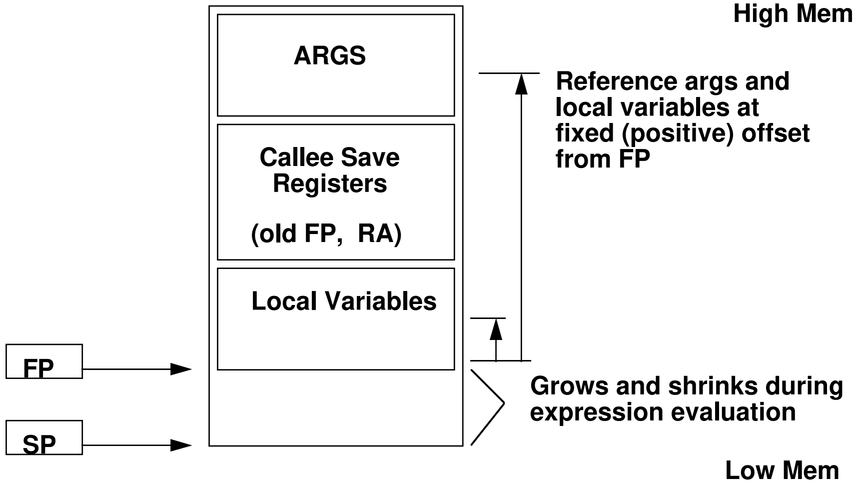
Stacks that Grow Up vs. Stacks that Grow Down:



How is empty stack represented?

<u>Little --> Big/Last Full</u>		<u>Little --> Big/Next Empty</u>	
POP:	Read from Mem(SP) Decrement SP	POP:	Decrement SP Read from Mem(SP)
PUSH:	Increment SP Write to Mem(SP)	PUSH:	Write to Mem(SP) Increment SP

Call-Return Linkage: Stack Frames



- Many variations on stacks possible (up/down, last pushed / next)
- Block structured languages contain link to lexically enclosing frame
- Compilers normally keep scalar variables in registers, not memory!

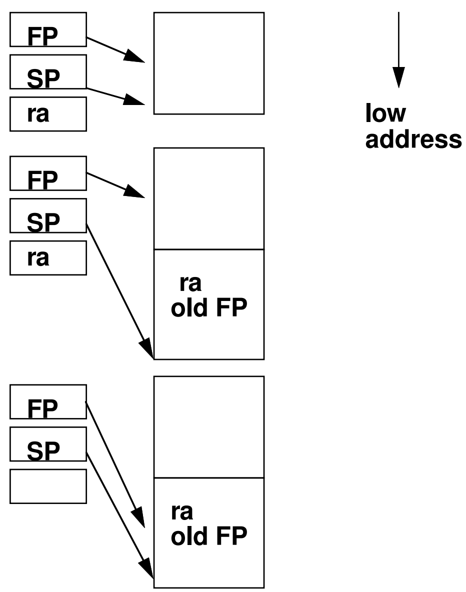
MIPS: Software conventions for Registers

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

Plus a 3-deep stack of mode bits.

MIPS / GCC Calling Conventions

```
fact:
  addiu    $sp, $sp, -32
  sw       $ra, 20($sp)
  sw       $fp, 16($sp)
  addiu    $fp, $sp, 32
. . .
  sw       $a0, 0($fp)
...
  lw       $31, 20($sp)
  lw       $fp, 16($sp)
  addiu    $sp, $sp, 32
  jr       $31
```



First four arguments passed in registers.

Details of the MIPS instruction set

- Register zero <u>always</u> has the value <u>zero</u> (even if you try to write it)
- Branch/jump <u>and link</u> put the return addr. PC+4 into the link register (R31)
- All instructions change <u>all 32 bits</u> of the destination register (including lui, lb, lh) and all read all 32 bits of sources (add, sub, and, or, …)
- Immediate arithmetic and logical instructions are extended as follows:
  - logical immediates ops are zero extended to 32 bits
  - arithmetic immediates ops are sign extended to 32 bits (including addu)
- The data loaded by the instructions lb and lh are extended as follows:
  - lbu, lhu are zero extended
  - lb, lh are sign extended
- Overflow can occur in these arithmetic and logical instructions:
  - add, sub, addi
  - it <u>cannot</u> occur in addu, subu, addiu, and, or, xor, nor, shifts, mult, multu, div, divu

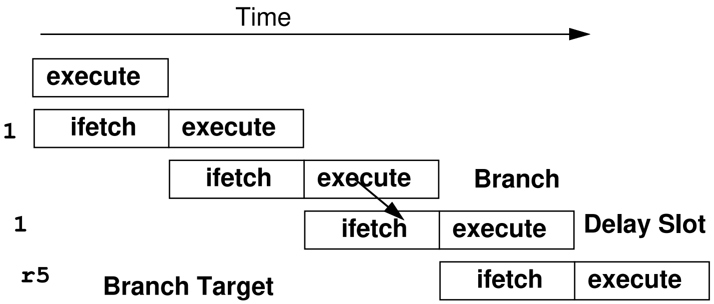
Delayed Branches

```
      li    r3, #7
      sub   r4, r4, 1
      bz    r4, LL
      addi  r5, r3, 1
      subi  r6, r6, 2
LL:   slt   r1, r3, r5
```

- In the “Raw” MIPS the instruction after the branch is executed even when the branch is taken?
  - This is hidden by the assembler for the MIPS “virtual machine”
  - allows the compiler to better utilize the instruction pipeline (???)

Branch & Pipelines

```
li r3, #7
sub r4, r4, 1
bz r4, LL
addi r5, r3, 1
LL: slt r1, r3, r5
```



By the end of Branch instruction, the CPU knows whether or not the branch will take place.

However, it will have fetched the next instruction by then, regardless of whether or not a branch will be taken.

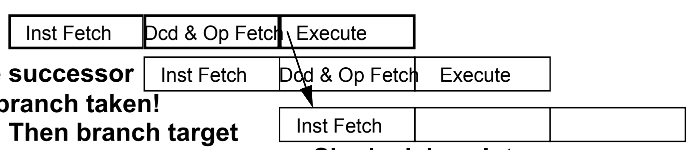
Why not execute it?

Filling Delayed Branches

Branch:

execute successor even if branch taken!

Then branch target or continue



Single delay slot impacts the critical path

- Compiler can fill a single delay slot with a useful instruction 50% of the time.
  - try to move down from above jump
  - move up from target, if safe

```
      add r3, r1, r2
      sub r4, r4, 1
      bz  r4, LL
      NOP
      ...
LL:   add rd, ...
```

Is this violating the ISA abstraction?

Miscellaneous MIPS I instructions

- break — A breakpoint trap occurs, transfers control to exception handler
- syscall — A system trap occurs, transfers control to exception handler
- coprocessor instrs. — Support for floating point
- TLB instructions — Support for virtual memory: discussed later
- restore from exception kernel/user — Restores previous interrupt mask & mode bits into status register
- load word left/right — Supports misaligned word loads
- store word left/right — Supports misaligned word stores

Performance

- Purchasing perspective
  - given a collection of machines, which has the
    - best performance ?
    - least cost ?
    - best performance / cost ?
- Design perspective
  - faced with design options, which has the
    - best performance improvement ?
    - least cost ?
    - best performance / cost ?
- Both require
  - basis for comparison
  - metric for evaluation
- Our goal is to understand cost & performance implications of architectural choices

Two notions of "performance"

Plane	DC to Paris	Speed	Passengers	Throughput (pmph)
Boeing 747	6.5 hours	610 mph	470	286,700
BAD/Sud Concodre	3 hours	1350 mph	132	178,200

Which has higher performance?

- **Time to do the task (Execution Time)**
  – execution time, response time, latency
- **Tasks per day, hour, week, sec, ns. .. (Performance)**
  – throughput, bandwidth

Response time and throughput often are in opposition

Definitions

- Performance is in units of things-per-second
  - bigger is better
- If we are primarily concerned with response time
  - performance(x) = $\frac{1}{\text{execution\_time}(x)}$

" X is n times faster than Y" means

$$n = \frac{\text{Performance}(X)}{\text{Performance}(Y)}$$

Example

- Time of Concorde vs. Boeing 747?
  - Concord is 1350 mph / 610 mph = 2.2 times faster
    = 6.5 hours / 3 hours
- Throughput of Concorde vs. Boeing 747 ?
  - Concord is 178,200 pmph / 286,700 pmph = 0.62 "times faster"
  - Boeing is 286,700 pmph / 178,200 pmph = 1.6 "times faster"
- Boeing is 1.6 times ("60%")faster in terms of throughput
- Concord is 2.2 times ("120%") faster in terms of flying time

We will focus primarily on execution time for a single job

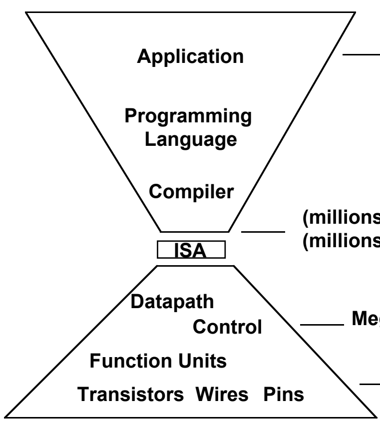
Basis of Evaluation

Pros		Cons
• representative	Actual Target Workload	• very specific • non-portable • difficult to run, or measure • hard to identify cause
• portable • widely used • improvements useful in reality	Full Application Benchmarks	•less representative
• easy to run, early in design cycle	Small “Kernel” Benchmarks	• easy to “fool”
• identify peak capability and potential bottlenecks	Microbenchmarks	• “peak” may be a long way from application performance

SPEC95

- **Eighteen application benchmarks (with inputs) reflecting a technical computing workload**
- **Eight integer**
  - **go, m88ksim, gcc, compress, li, ijpeg, perl, vortex**
- **Ten floating-point intensive**
  - **tomcatv, swim, su2cor, hydro2d, mgrid, applu, turb3d, apsi, fppp, wave5**
- **Must run with standard compiler flags**
  - **eliminate special undocumented incantations that may not even generate working code for real programs**

Metrics of performance



Answers per month
Useful Operations per second

(millions) of Instructions per second – MIPS
(millions) of (F.P.) operations per second – MFLOP/s

Megabytes per second

Cycles per second (clock rate)

Each metric has a place and a purpose, and each can be misused

Aspects of CPU Performance

$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

	instr. count	CPI	clock rate
Program			
Compiler			
Instr. Set Arch.			
Organization			
Technology			

CPI

"Average cycles per instruction"

CPI = (CPU Time * Clock Rate) / Instruction Count
= Clock Cycles / Instruction Count

$$\text{CPU time} = \text{ClockCycleTime} * \sum_{i=1}^{n} \text{CPI}_i * I_i$$

$$\text{CPI} = \sum_{i=1}^{n} \text{CPI}_i * F_i \quad \text{where} \quad F_i = \frac{I_i}{\text{Instruction Count}}$$

"instruction frequency"

Invest Resources where time is Spent!

Example (RISC processor)

Base Machine (Reg / Reg)

Op	Freq	Cycles	CPI(i)	% Time
ALU	50%	1	.5	23%
Load	20%	5	1.0	45%
Store	10%	3	.3	14%
Branch	20%	2	.4	18%
			2.2	

Typical Mix

How much faster would the machine be is a better data cache reduced the average load time to 2 cycles?

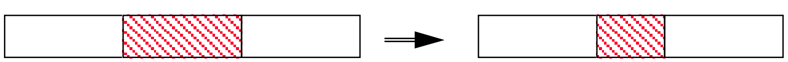
How does this compare with using branch prediction to shave a cycle off the branch time?

What if two ALU instructions could be executed at once?

Amdahl's Law

Speedup due to enhancement E:

$$\text{Speedup(E)} = \frac{\text{ExTime w/o E}}{\text{ExTime w/ E}} = \frac{\text{Performance w/ E}}{\text{Performance w/o E}}$$



Suppose that enhancement E accelerates a fraction F of the task

by a factor S and the remainder of the task is unaffected then,

ExTime(with E) $\leq$ ((1-F) + F/S) X ExTime(without E)

$$\text{Speedup(with E)} \leq \frac{1}{(1-F) + F/S}$$

Summary: Salient features of MIPS I

- **32-bit fixed format inst** (3 formats)
- **32 32-bit GPR** (R0 contains zero) and 32 FP registers (and HI LO)
  - partitioned by software convention
- **3-address, reg-reg arithmetic instr.**
- **Single address mode for load/store:** base+displacement
  - –no indirection, scaled
- –**16-bit immediate plus LUI**
- **Simple branch conditions**
  - compare against zero or two registers for =,≠
  - no integer condition codes
- **Delayed branch**
  - execute instruction after the branch (or jump) even if the branch is taken (Compiler can fill a delayed branch with useful work about 50% of the time)

Summary: Instruction set design (MIPS)

- Use general purpose registers with a load-store architecture: <u>YES</u>
- Provide at least 16 general purpose registers plus separate floating-point registers: <u>31 GPR & 32 FPR</u>
- Support basic addressing modes: displacement (with an address offset size of 12 to 16 bits), immediate (size 8 to 16 bits), and register deferred; : <u>YES: 16 bits for immediate, displacement (disp=0 => register deferred)</u>
- All addressing modes apply to all data transfer instructions : <u>YES</u>
- Use fixed instruction encoding if interested in performance and use variable instruction encoding if interested in code size : <u>Fixed</u>
- Support these data sizes and types: 8-bit, 16-bit, 32-bit integers and 32-bit and 64-bit IEEE 754 floating point numbers: <u>YES</u>
- Support these simple instructions, since they will dominate the number of instructions executed: load, store, add, subtract, move register-register, and, shift, compare equal, compare not equal, branch (with a PC-relative address at least 8-bits long), jump, call, and return: <u>YES, 16b</u>
- Aim for a minimalist instruction set: <u>YES</u>

Summary: Evaluating Instruction Sets?

Design-time metrics:

- Can it be implemented, in how long, at what cost?
- Can it be programmed? Ease of compilation?

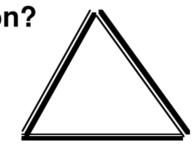
Static Metrics:

- How many bytes does the program occupy in memory?

Dynamic Metrics:

- How many instructions are executed?
- How many bytes does the processor fetch to execute the program?
- How many clocks are required per instruction?
- How "lean" a clock is practical?

***Best Metric*:** <u>Time to execute the program!</u>



NOTE: this depends on instructions set, processor organization, and compilation techniques.