Lecture 3: The Instruction Set Architecture (cont.)

COS / ELE 375

Computer Architecture and Organization

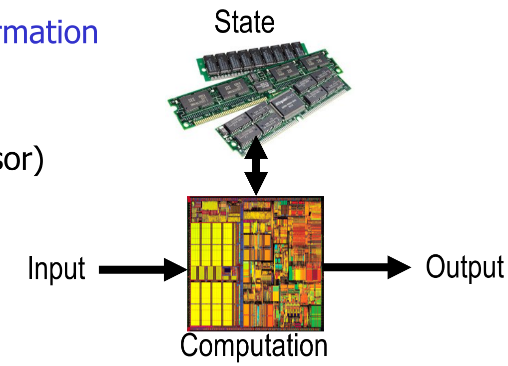
Princeton University
Fall 2015

Prof. David August

Review: Instructions

Computers process information

- Input/Output (I/O)
- State (memory)
- Computation (processor)



- Instructions instruct processor to manipulate state
- Instructions instruct processor to produce I/O in the same way

Review: State

Typical modern machine has this architectural state:

- Main Memory
- Registers
- Program Counter

Architectural – Part of the programmer's interface
(implementation likely to have additional state)

Review: Instructions

An ADD Instruction:

add r1 = r2 + r3 (assembly)

Opcode Operands

Parts of the Instruction:

- Opcode (verb) – what operation to perform
- Operands (noun) – what state to manipulate
- Source Operands – where values come from
- Destination Operand – where to deposit data values

Review: Instructions

36: add r1 = r2 + r3
40: sub r4 = r5 + r6
44: load r7 = M[r8]
48: store M[r9] = r10
48: branch r11 > 0, 56
52: jump 36
56: halt

Topics For Today

- Function Calls and Calling Convention
- Big and Little Endian
- Addressing Modes
- Pseudo-ops
- Instruction Set Variety
- RISC vs. CISC

Function Calls

Recall Our Example:

10: “Hello\n” ; data in memory
36: arg1 = 10 ; argument memory address is 10
40: r1 = 10
44: r1 = r1 - 1
48: call printf ; printf(arg1)
48: branch r1 > 0, 44
52: halt

Cheesy Function Demo

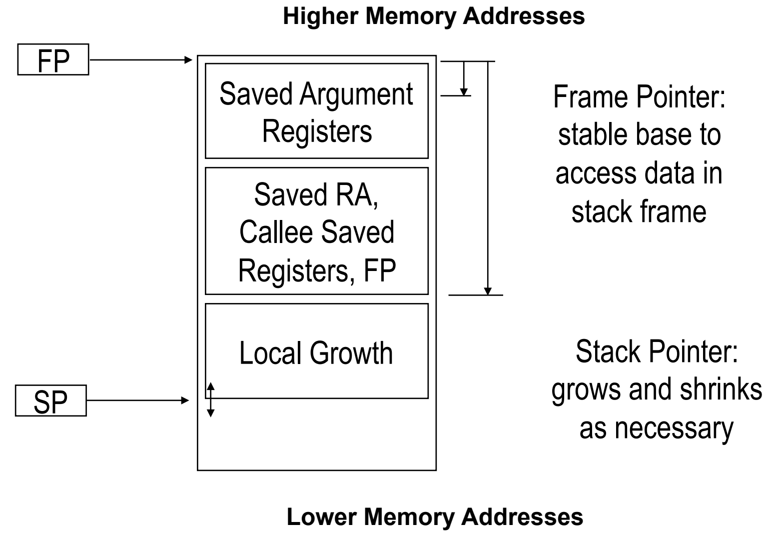
What state must be passed to function?
What state must be passed back from function?
What state must be preserved across call?
Whose responsibility is it?

Calling Convention

- Calling convention - standard defined for architecture
- Register component of calling convention:

Name	Register number	Usage
$zero	0	the constant value 0
$v0-$v1	2-3	values for results and expression evaluation
$a0-$a3	4-7	arguments
$t0-$t7	8-15	temporaries
$s0-$s7	16-23	saved
$t8-$t9	24-25	more temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address

Calling Convention: Stack Frames



The Life of a Function Call

1. Push to stack important values in temp registers: caller saved ($t0, $t9)
2. Place the return address in agreed upon reg/stack ($ra)
3. Place parameters in agreed upon reg/stack ($a0-$a3)
4. Jump to procedure
    1. Allocate new stack frame ($fp, $sp)
    2. Push registers: callee saved ($s0-$s7, $ra, old $fp)
    3. Do the work of the procedure
    4. Place return value in agreed upon location ($v0, $v1)
    5. Pop callee saved values ($s0-$s7, $ra, old $fp)
    6. Deallocate stack frame ($fp, $sp)
    7. Return to return address
5. Restore caller saved values ($t0-$t9)
6. Continue

Implementing a Function Call

- Special call instruction on processor does some work
- Alternatively, program does all the work
    - Use store, load, and move instructions to save data
    - Use control flow instructions to jump to the function
    - MIPS has no call instruction
    - Push $s1 and $s2:

      M[$sp - 4] = $s1
      M[$sp - 8] = $s2
      $sp = $sp - 8
- Either way, calling convention must be respected

Memory Addressing

View memory as a one-dimensional array

1980+: Elements of array are 8-bits

We say "byte addressable"

Assuming 32-bit words:

Address	Data
0	8 bits of data
1	1 byte of data
2	2 nibbles of data
3	¼ word of data
…	…
FFFFFFFF	8 bits of data

1. Can a word start at any address?
   MIPS: no, aligned, 2 Least Significant Bits (LSB) are 0
   x86: yes

2. How are bytes of word laid out?
   MIPS/IA-64: Big or Little Endian
   x86: Little Endian ?

Endian

3	2	1	0

- Little Endian: least-significant byte is stored in the location with the lowest address (little end first)

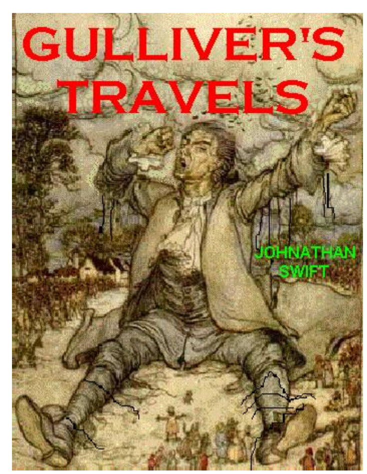
Address	0000	0001	0002	0003
Byte #	0	1	2	3

- Big Endian: most-significant byte is stored in the lowest address (big end first)

Address	0000	0001	0002	0003
Byte #	3	2	1	0

Endian Origin

- *Gulliver's Travels* by Jonathan Swift
  Two countries go to war over which end of soft-boiled egg should be eaten first - the big or little end



What does this program do?

```
#include <stdio.h>

main() {
  unsigned int a;
  unsigned char *p;

  a = 0xabcd1234;
  p = (unsigned char *) &a;

  fprintf(stdout, “%x\n”, *p);
}
```


Addressing Modes

- Data/state for instructions can be anywhere:
  - In memory, encoded in instructions
  - In memory data area
  - In registers
- Data an be specified in many ways:

  load r1 = M[0] → r1 = DEADBEEF

  load r1 = M[r0] → r1 = DEADBEEF

  load r1 = M[r0 + 1] → r1 = 0

  load r1 = M[M[1]] → r1 = DEADBEEF

Address	Data
0	DEADBEEF
1	0
2	471A471B

Immediates

- Small constants are used quite frequently (50% of operands)
  e.g., A = A + 5;
  B = B + 1;
  C = C - 18;
- Options:
  1. Put 'typical constants' in binary/memory and load them.
  2. Create hard-wired registers (like $0/$zero) for constants.
  3. Immediates: Encode constants in the instruction
- MIPS Instructions:

```
addi $29, $29, 4
slti $8, $18, 10
andi $29, $29, 6
ori $29, $29, 4
```

Register and Direct Addressing

- Direct/Absolute Addressing: address is immediate
  Example:
  load r1 = M[1500]
- Register: register number is immediate
- Useful for addressing locations fixed during execution
  - Branch target addresses
  - C global variable and static variable memory locations

Register Indirect and Displacement Addressing

- Register Indirect Addressing: address from register
  Example:
  load r1 = M[r2]

  Useful for pointers
- Displacement: register + immediate
  Example:
  load r1 = M[r2 + 100]

  Useful for addressing locations in static array
  Useful for structs on heap (dynamically allocated)

Register Indirect and Displacement Addressing

- Index/Base Register Addressing: 2 registers
  Example:
  load r1 = M[r2 + r3]

  Useful for accessing dynamic offsets into dynamic structs/arrays
- Memory Indirect Addressing: address in memory
  Example:
  load r1 = M[M[r2]]
  load r1 = M[M[2000]]

  Useful for dereferencing pointers in structs such as next fields in linked list

PC-Relative

- PC-Relative: like base + displacement, implied base
  Allows longer displacement immediate, why?

  Used for branch instructions:
  jump [- 8] ; jump back 2 instructions

  Assembly uses labels:
  FooBar:
  add r1 = r2 + r3
  jump FooBar

  Assembler or linker determine actual the immediate…

Other, Crazy Modes

- Scaled: address from registers/immediates, some scaled

  Example:

  load r1 = M[100 + r2 + r3 * d]

  d is defined by instruction

- You are bound to see others
- Make up your own…

Addressing Modes

Immediate	add r1 = r2 + 5
Register	add r1 = r2 + r3
Direct	load r1 = M [4000]
Register Indirect	add r1 = r2 + M[r2]
Displacement	load r1 = M[r2 + 4000]
Indexed/Base	add r1 = r3 + M[r2 + r3]
Memory Indirect	load r1 = M[M[r2]]
PC Relative	branch r1 < r3, 1000
Scaled	load r1 = M[100 + r3 + r4 * d]

Memory Addressing Mode Usage? (ignore immediate/register mode)

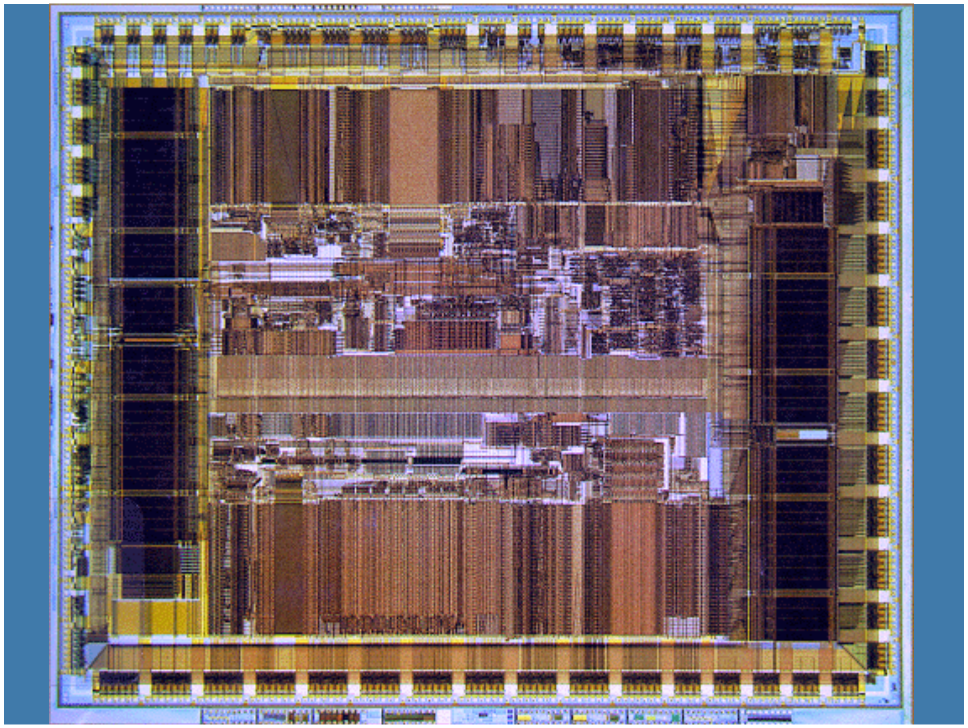
<u>A few programs measured:</u>

Displacement:	42% avg, 32% to 66%
Direct:	33% avg, 17% to 43%
Register Indirect:	13% avg, 3% to 24%
Scaled:	7% avg, 0% to 16%
Memory Indirect:	3% avg, 1% to 6%
Other:	2% avg, 0% to 3%

75% Displacement + Direct

88% Displacement + Direct + Register Indirect

Optimizations…



Review: MIPS Instruction Set

- MIPS – SGI Workstations, Nintendo, Sony…

State:

- 32-bit addresses to memory (32-bit PC)
- 32 32-bit Registers
- A “word” is 32-bits on MIPS
- Register $0 ($zero) always has the value 0
- By convention, certain registers are used for certain things – more next time…

31

Review: MIPS Instruction Set

- Add: add $t1 = $t2 + $t3
- Subtract: sub $t1 = $t2 + $t3
- Load Word: lw $t1, 100 ($t2)
- Store Word: sw $t1, 100 ($t2)
- Jump: j 100

- Branch Not Equal: bne $t1, $t2, 100
- Branch Equal: beq $t1, $t2, 100

- Why no “blt” instruction?

32

Control Flow

- Branch changes the flow of instructions through the processor
- We say that branch instructions are “control flow instructions”
- Control Flow: test values to make decisions

- Example:

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

```
     beq $s4, $s5, Label1
     add $s3, $s4, $s5
     j Label2
Label1:
     sub $s3, $s4, $s5
Label2:
     ...
```

Control Flow

- Why no blt?
- New instruction:

```
                       if  $s1 < $s2 then
                              $t0 = 1
slt $t0, $s1, $s2           else
                              $t0 = 0
```

- Can use this instruction to build a "blt”
- Assembler has “blt”, but assembler needs a free register to hold temporary value.
- Assembler uses Register Convention just like in calling convention

MIPS Encodings
32-bits/Instruction

	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
R:	op	rs	rt	rd	shamt	funct
I:	op	rs	rt	address / immediate		
J:	op	target address				

op: basic operation of the instruction (opcode)
rs: first source operand register
rt: second source operand register
rd: destination operand register
shamt: shift amount
funct: selects the specific variant of the opcode (function code)
address: offset for load/store instructions ($+/-2^{15}$)
immediate: constants for immediate instructions

MIPS Immediates

- Formats:

I	op	rs	rt	16 bit address
J	op	26 bit address		

- Addresses are 32 bits, immediates are not!
- How do control flow instructions handle this?
- How do we handle this with load and store instructions?

MIPS:

MIPS operands

Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS:

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`	Three operands; data in registers
	subtract	`sub $s1, $s2, $s3`	`$s1 = $s2 - $s3`	Three operands; data in registers
	add immediate	`addi $s1, $s2, 100`	`$s1 = $s2 + 100`	Used to add constants
Data transfer	load word	`lw $s1, 100($s2)`	$s1 = Memory[$s2 + 100]	Word from memory to register
	store word	`sw $s1, 100($s2)`	Memory[$s2 + 100] = $s1	Word from register to memory
	load byte	`lb $s1, 100($s2)`	$s1 = Memory[$s2 + 100]	Byte from memory to register
	store byte	`sb $s1, 100($s2)`	Memory[$s2 + 100] = $s1	Byte from register to memory
	load upper immediate	`lui $s1, 100`	$\$s1 = 100 * 2^{16}$	Loads constant in upper 16 bits
Conditional branch	branch on equal	`beq $s1, $s2, 25`	if (`$s1 == $s2`) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1, $s2, 25`	if (`$s1 != $s2`) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1, $s2, $s3`	if (`$s2 < $s3`) `$s1` = 1; else `$s1` = 0	Compare less than; for beq, bne
	set less than immediate	`slti $s1, $s2, 100`	if (`$s2 < 100`) `$s1` = 1; else `$s1` = 0	Compare less than constant
Unconditional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to `$ra`	For switch, procedure return
	jump and link	`jal 2500`	`$ra` = PC + 4; go to 10000	For procedure call

Assembly vs. Machine Language

- Assembly provides convenient symbolic representation
  - Make it readable to that feeble race of humans
  - Text, destination first, single opcode
- Machine language is the underlying reality
  - Bits/Numbers
  - MIPS: destination in middle, opcode split
- Assembly can provide 'pseudo-ops'
  - Example:
    “move $t0, $t1”

    Can be implemented using “add $t0, $t1, $zero”
- For performance, examine real instructions

List of Instruction Sets

- Sun Microsystem’s SPARC
- IBM/Motorola’s PowerPC
- Hewlett-Packard’s PA-RISC
- Intel’s x86
- DEC Alpha
- Motorola’s 68xxx
- Intel’s IA-64
- SGI’s MIPS
- ARM
- SuperH
- TI’s C6x
- …

How are they different?

- ISA Class
- Assembly
- Encodings
- RISC vs. CISC
- State
- Addressing Modes

Different answers a result of different design goals!

Basic ISA Classes

Accumulator (1 register):

"add A": accumulator = accumulator + mem[A]

Stack (Think HP calculator):

"add": push(pop() + pop())

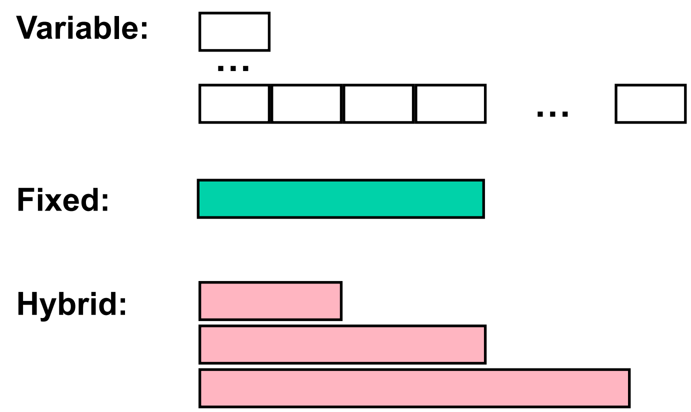
General Purpose Register, Register-Memory:

"add r1, M[r2]": r1 = r1 + M[r2]

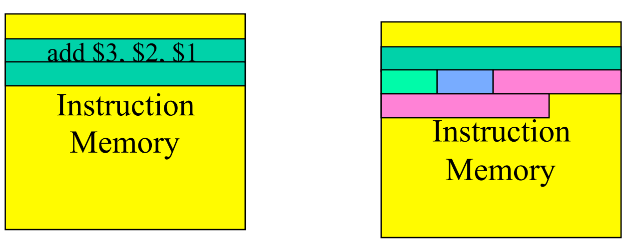
GPR, Load/Store (AKA Register-Register): MIPS

"add r1 r2 r3": r1 = r2 + r3

Instruction Encoding Widths



Instruction Widths



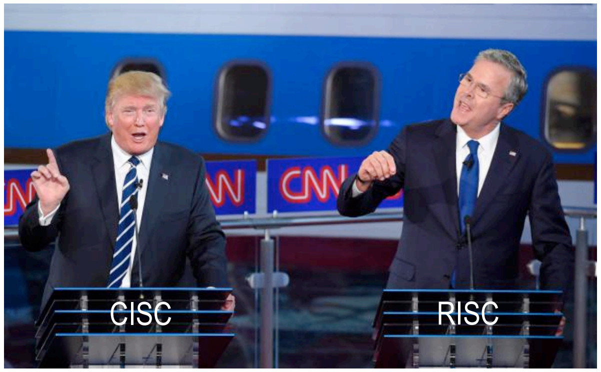
- MIPS & ARM: Fixed instruction width (32 bits)
- Variable instruction width impact on implementation?

State and Addressing Modes

- Register Count
  - ~8 Integer Registers: x86
  - 32 Integer Registers: MIPS
  - 128 Integer Registers: IA-64
  - What size is ideal?
- Addressing Modes
  - Lots - x86
  - A Few - MIPS, IA-64
  - Which is better?

Great Debate: CISC vs. RISC

- CISC: Complex Instruction Set Computer
- RISC: Reduced Instruction Set Computer



The RISC Design Philosophy

KEEP DESIGN SIMPLE!

- Keep number of instruction types small
- Use easier to manipulate fixed length instructions
- Do simple instructions faster
- Use only a few key addressing modes

The CISC Design Philosophy

MAKE MACHINE EASY TO PROGRAM!

- Support for frequent tasks
  - Functions: Provide a "call" instruction, save registers
  - Strided Array Access: Provide special addressing mode
- Make each instruction do lots of work
  - Less explicit state necessary
  - Fewer instructions necessary

Arguments

- RISC
  - Clearly superior, that is why they are covered in the textbook
  - Load/Store architectures dominate new architectures
  - Easier to add advanced performance enhancements
  - Smaller design teams necessary
- CISC
  - Binaries are smaller (x86 is 20% smaller than MIPS)
  - Machines are faster in practice
  - Almost all processors used in desktop computers are CISC
  - Clearly the winner, because you probably use one now

What do you think? Who is winning or who has won?

List of Instruction Sets

- Sun Microsystem’s SPARC
- IBM/Motorola’s PowerPC
- Hewlett-Packard’s PA-RISC
- Intel’s x86
- DEC Alpha
- Motorola’s 68xxx
- Intel’s IA-64
- SGI’s MIPS
- ARM
- SuperH
- TI’s C6x
- …

List of Successful Instruction Sets

- Intel’s x86

This lecture was brought to you by Intel Corporation.

x86: A Dominant Architecture

- See your textbook for a more detailed description

- Complexity:
  - Instructions from 1 to 17 bytes long
  - One operand must act as both a source and destination
  - One operand can come from memory
  - Complex addressing modes
    Example: “base or scaled index with 8 or 32 bit displacement”

- Saving grace:
  - The most frequently used instructions are not too difficult to build
  - Compilers avoid the portions of the architecture that are slow

“what the 80x86 lacks in style is made up in quantity, making it beautiful from the right perspective” -- unknown

x86: An Evolving Architecture

- 1978: Intel announces 8086 (16 bit architecture)
- 1980: 8087 floating point coprocessor is added
- 1982: 80286 increases address space to 24 bits, new instructions
- 1985: 80386 extends to 32 bits, new addressing modes
- 1989-1995: 80486, Pentium, Pentium Pro add instructions
- 1997: MMX is added

“This history illustrates the impact of the “golden handcuffs” of compatibility”

“adding new features as someone might add clothing to a packed bag”

“an architecture that is difficult to explain and impossible to love”

Summary and Next Time

Summary:

- Function Calls and Calling Convention
- Big and Little Endian
- Addressing Modes
- Pseudo-ops
- Instruction Set Variety
- RISC vs. CISC

- Next Time: Performance!
- Read: Chapters 1, 2, and 3