ECE-425: Lecture 3 Instruction Set Architecture

Prof: Mohamed El-Hadedy

Email: mealy@cpp.edu

Office: 909-869-2594

Instructions

- Language of the Machine
- More primitive than higher level languages
- Redefine the instruction set until you see the real computer language
- Few basic operations all machines should provide.
- Similar operations, similar languages

Instructions

- We’ll be using the MIPS instruction set architecture
- MIPS: Microprocessor without Interlocked Pipeline Stages
- MIPS: is RISC architecture
- MIPS: was developed as a pat of a VLSI research program at Stanford University in the early 80s.
- Used by NEC, Nintendo, Silicon Graphics, Sony

Common goal: Find the language that makes it easy to build HW and compiler while maximizing performance and minimizing cost

Instructions

- Interlock Pipeline: Hardware is used to check for hazards between stages
- Non interlocked pipeline: no hardware is used to check for hazards.
- RISC: the idea of leaving everything to software (adding flexibility to the architecture).
- A typical RISC machine, 5.5M MIPS chips sold, used in computers, printers, network cards, video game (Nintendo).
- Later versions of the MIPS ISA did have interlocking ☺

Instruction Types

- What types of instructions should be included in a general purpose processor instruction set?
- Should satisfy:
  - Complete (able to develop programs to evaluate computable functions *reasonably*)
  - Efficient (frequently used functions available)
  - Regular (expected opcodes & addressing modes)
  - Compatible (with existing machine family- as possible)

Completeness of Instruction Set

- Should include the following five types:
  - Arithmetic instructions
  - Data transfer
  - Logic Instructions
  - Program-control instructions
  - Input-output

MIPS Architecture (MIPS Instructions)

- The MIPS instruction set consists of about 111 total instructions, 32-bit RISC Processor
- Programmable storage: 32 32-bit general purpose registers (GPR), represented as $0..$31, the content of $0 is always 0.
- Pipelined Execution of Instructions
- All instructions are 32-bits
- Most Instructions executed in one clock cycle
- 2^{30} 32-bit memory words, such as Memory[0], Memory[8], etc
  - ✓Memory holds data structures such as arrays, and spilled registers.

RISC vs CISC

- RISC (reduced instruction set computer) instructions
  - only load/store instructions access memory
  - data (i.e., operands) must be in registers to perform operation
  - each instruction roughly taking same amount of time
  - simple addressing modes
  - virtually all new instruction sets since 1982 have been RISC (M 68000 announced in year 1980)
  - etc.
- CISC (complex instruction set computer) instructions
  - alu/logic instructions access memory to fetch operands
  - load/store instructions access memory
  - some instructions' execution time is much longer than other instructions
  - complex addressing modes
  - etc.

MIPS Instructions

- Byte addressable memory
  - A 32-bit word contains four bytes
  - To address the next word of memory add 4
- 32 32-bit floating point registers, paired for double precision
- What is a spilled register?
  - A variable is less commonly used and saved in memory rather than in registers

MIPS Arithmetic

- All instructions have 3 operands
- Operand order is fixed (destination first)

  Example:

  C code: A = B + C

  MIP code: add $s0, $s1, $s2

  (associated with variables by compiler)

- MIPS compiler uses $s0, $s1, … for registers that correspond to variables in C program and $t0, $t1, …

  for temporary registers needed to compile a program

MIPS Arithmetic

- Another Example

```
C code: A = B + C + D;
              E = F - A;

MIPS code:    add $t0, $s1, $s2        #comments here
              add $s0, $t0, $s3        #comments here
              sub $s4, $s5, $s0        #comments here
```

- Operand must be registers, only 32 registers provided
- Design Principle: smaller is faster. Why ?

MIPS Arithmetic (Cont.)

Ex: f = (g + h) – (i + j);

f, g, h, i and j assigned to $s0 to $s4

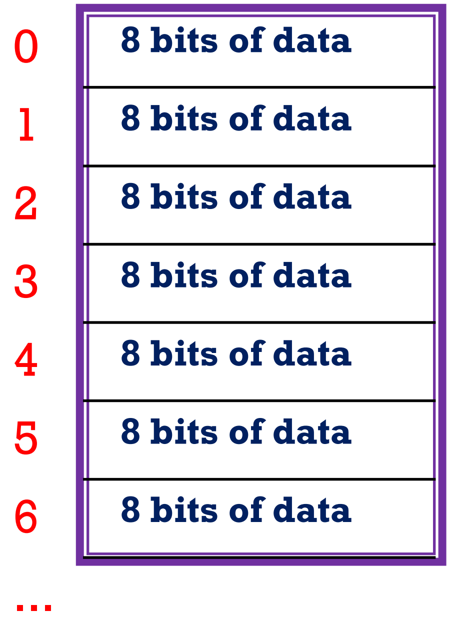
```
add  $t0, $s1, $s2
add  $t1, $s3, $s4
sub  $s0, $t0, $t1
```

MIPS Instructions

- MIPS arithmetic instructions
    - add, addi, addu, sub, subi, subu, addiu
    - all arithmetic operations operate on words
    - ADD: adds the value in two registers (with overflow)
    - ADDi: adds an immediate value (constant) to the register
    - ADDU: add unsigned (no overflow)
    - SUB: subs the value in two registers
    - SUBI: subs an immediate value to the register
    - SUBU: subs unsigned (no overflow)
    - ADDIU: add immediate unsigned (no overflow)

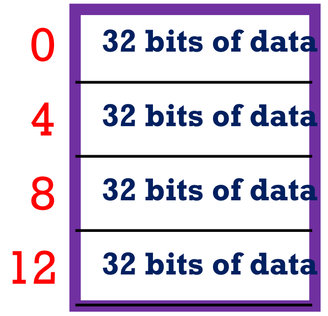
Memory Organization

- Viewed as a large, single-dimension array, with an address.
- A memory address is an index into the array
- "Byte addressing" means that the index points to a byte of memory.



Memory Organization

- Most data items use larger "words“ not bytes
- For MIPS, a word is 32 bits or 4 bytes.



Registers hold 32 bits of data

- 2^{32} bytes with byte addresses from 0 to 2^{32}-1
- 2^{30} words with byte addresses 0, 4, 8, ... 2^{32}-4
- Words are aligned

  i.e., what are the least 2 significant bits of a word address?

MIPS Addressing Modes

- Register - Uses value in register as operand
  - Example $2 - Uses register 2 as operand
- Direct Address - Uses value stored in memory at given address
  - Example 100 - Uses value stored at location 100 in memory as operand
- Register Indirect Address - Uses value in register as address of operand in memory
  - Example ($3) - Uses value in register 3 as address of memory operand

Memory Instructions

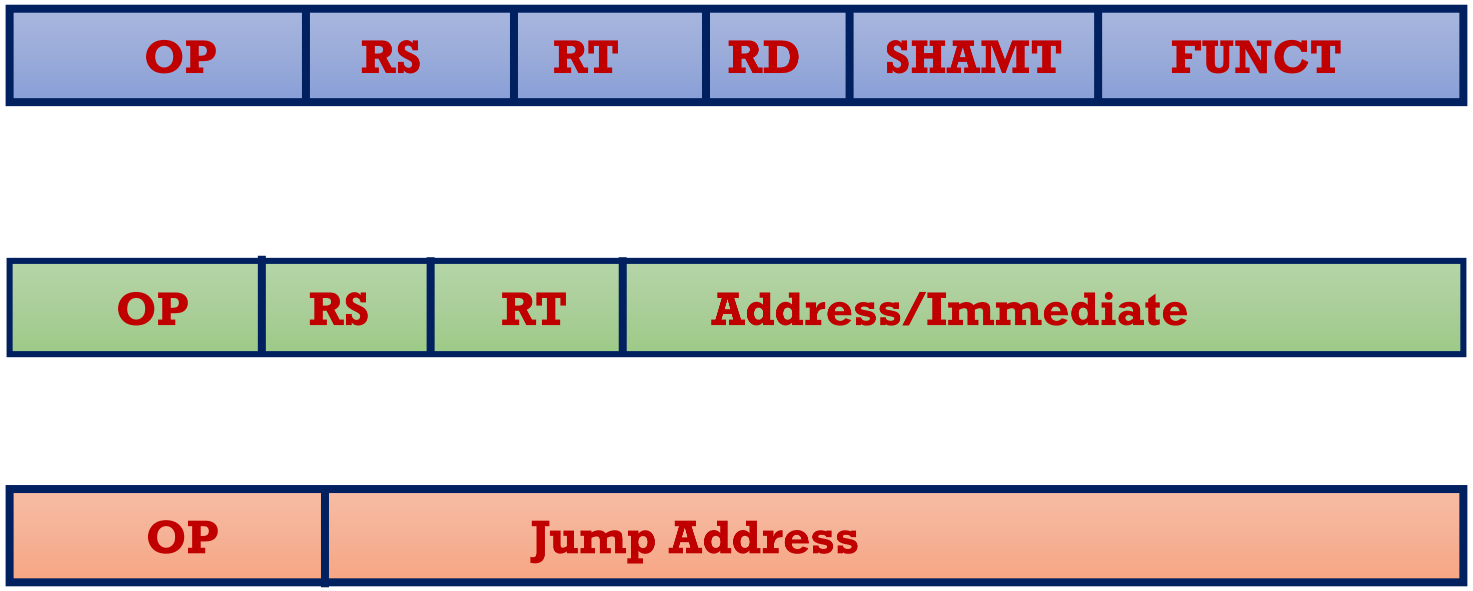
- Load Instructions:
  - ✓Instruction: LBU: Load Byte Unsigned
  - ✓Instruction: LHU: Load half-word unsigned
  - ✓Instruction: LW: Load word
- Store Instructions:
  - ➢Instruction: SB: Store Byte
  - ➢Instruction: SH: Store half-word
  - ➢Instruction: SW: store word

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 Instruction Formats

- 32-bit Instruction Formats R, I and J



MIPS Instruction Formats

- **R Instructions:** are used when all the data values used by the instruction are located in registers.
- **Example:** OP rd, rs, rt , when OP is the mnemonic for the particular instruction. RS and RT are the source registers, and RD is the destination register.
- **R Format:**

opcode	RS	RT	RD	Shift (shmat)	Funct
6-bits	5-bits	5-bits	5-bits	5-bits	6- bits

- **Shift(shmat):** used with the shift and rotate instructions, this is the amount by which the source operand RS is rotated/shifted. This field is 5-bits long (6 to 10).
- **Funct:** For the instructions that share an opcode, the funct parameter contains the necessary control codes to differentiate the different instructions. Example: Opcode 0x00 access the ALU, and the funct selects which ALU function to use (lets say 0x20 refers to adding specifically)

MIPS Instruction Formats

- **I Instructions:** are used when the instruction must operate on an immediate value and a register value.
- Immediate values may be a maximum of 16-bits long.
- **I Format:**

OpCode	RS	RT	IMM
6-bits	5-bits	5-bits	16-bits

- **Opcode:** 6-bit opcode of the instruction. In I instruction, all mneumonics have a one –to-one correspondence with the underlying opcodes.
- **RS, RT:** The source and target register operands, respectively. 5-bits each (21 to 25 and 16 to 20, respectively)
- **IMM:** The 16-bit immediate value. 16-bits (0 to 15). This value is used as the offset in various instructions, and depending on the instruction, may be expressed in two's complement.
- **Example:** **andi $1, $2, 100; immediate = 100, rs = $2, rt=$1; op=12; $1=$2 & 100**

MIPS Instruction Formats

- **J Instructions:** are used when a jump needs to be performed. The J instruction has the most space for an immediate value, because addresses are large numbers.
- **Example:** NASA (label) OP, when OP is the mnemonic for the particular jump instruction, and the label is the target address to jump to (NASA in this example is the label).
- **J Format:**

OPcode	**Pseudo-Address**

- **Opcode:** The 6-bit opcode corresponding to the particular jump command
- **Address:** A 26-bit shortened address of the destination. The two least significant bits are removed, and the 4 most significant bits are removed, and assumed to be the same as the current instruction's address.

MIPS Instructions

- J type instruction

	op	target address
no. of bits	6	26

jump instruction;

j 10000; go to target address 10000
jal 10000; $31 = PC + 4; go to 10000; for procedure call,
; return address is saved in $31

MIPS Instructions

- Instructions for making decisions
  - alter the control flow,
  - change the “next” instruction to be executed

beq \$1, \$2, L1; (I type)

- go to the instruction labeled L1, if the value in \$1 equals the value in \$2

bne \$3, \$4, L1; (I type)

- go to the instruction labeled L1, if the value in \$3 not equal the value in \$4

- **example:** C code if (i==j) h = i+j;

MIPS code

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

MIPS Instructions

- Control
  - MIPS unconditional branch instructions:

```
j         label
```

Example:

```
if(i != j)                    beq $s4, $s5, Lab1
   h = i + j;                         add $s3, $s4, $s5
else                                  j Lab2
   h = i - j;                         Lab1: sub $s3, $s4, $s5
                                      Lab2: ............
```

- Other cases such as compiling a while loop, a switch statement, a loop with array index, a procedure call, nested procedures into MIPS code (skip, this is too much into assembly language programming)
- This course is to design a processor

MIPS Instruction Formats

- **FR Instructions:** FR instructions are similar to the R instructions, except they are reserved for use with floating-point numbers:

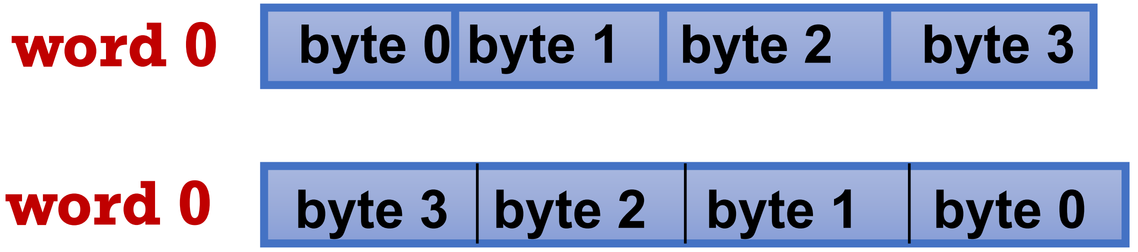
opcode	FMT	FT	FD	Funct

- **FI Instructions:** FI instructions are similar to the I instructions, except they are reserved for use with floating-point numbers

OpCode	FMT	FT	IMM
6-bits	5-bits	5-bits	16-bits

Addressing Objects

- Big Endian: address of most significant byte = word address, (xxx…00 = big end of the word)
  - IBM, Motorola 68K, MIPS, Sparc, HP PA
- Little Endian: address of least significant byte = word address, (xxx…00 = little end of the word)
  - Intel 80x86, DEC Vax, DEC Alpha
- alignment : require that objects fall on address that is multiple of their size



Byte-Index	0	1
Big-Endian	12	34
Little-Endian	34	12

Data Types

- Bit
- Bit String:
  - 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
- Decimals : digits 0-9 encoded as 0000 through 1001
- Integers : 2's complement
- Floating Point:
  - single precision, double precision

MIPS Instructions

- MIPS arithmetic instructions
    - add, addi, addu, sub, subi, subu, addiu
    - all arithmetic operations operate on words

Which add for address arithmetic ?

Which add for integers ?

MIPS Instructions

- Logical Instructions
  - and, or, xor, nor, andi (and immediate), ori (or immediate), sll (shift left logical), srl (shift right logical), sra (shift right arithmetic),
  - examples
    - sll $1, $2, 10 ; $1 = $2 << 10 (shift left logical 10 bits of $2)
    - srl $1, $2, 10 ; $1 = $2 >> 10 (shift right logical 10 bit)
    - or $1, $2, $3 ; $1 = $2 | $3
    - and $1, $2, $3 ; $1 = $2 & $3
    - ori $2, $3, 99 : $2 = $3 | 99

MIPS Instructions

- data transfer instructions

lw	load word
sw	store word
lbu	load byte unsigned
sb	store byte
lui	load upper immediate

  - lui \$1, 100 ; \$1 = 100 x 2^{16}
  - load 100 into the upper 16 bits of \$1,

MIPS Instructions

- slt set on less than, compare 2's comp.
  - slt $1, $2, $3; if $2 < $3 then $1 = 1 else $1 = 0.
- slti set on less than immediate, compare 2's comp.
  - slti $2, $3, 79; if $3 < 79 then $2 = 1, else $2 = 0
- sltu set on less than unsigned, compare natural numbers
- sltiuset on less than immediate unsigned, compare natural numbers
- jr jump register,
  - jr $31 ; go to register $31

MIPS Instructions

- Signed vs Unsigned Comparison

$1 = 0000 0000 0000...... 00001

$2 = 1111 1111 1111....... 11111

$3 = 0000 0000 0000........ 00000

slt $4, $1, $2

sltu $5, $1, $2

slt $6, $2, $3

sltu $7, $2, $3

$4=_______________; $5 = ________________

$6=_______________; $7 = _________________

MIPS Instructions

- $0 is always zero (you can not change it)
- branch and link save the return address PC+4 into $31
- all instructions change all 32 bits of the destination register
- logic immediates are zero extended to 32 bits
- arithmetic immediates are sign extended to 32 bits
- data loaded by the instructions lb, lh are extended as follows:
  - lbu, lhu are zero extended (unsigned load)
  - lb, lh are sign extended
- overflow can occur in signed arithmetic/logic instructions, will not occur in unsigned arithmetic/logic instructions.

Using Variable Array Index

Ex: g = h + A[i];   g, h, i are in $s1, $s2, $s4 base register is $s3

To get the word address you need i*4

```
add   $t1, $s4, $s4
add   $t1, $t1, $t1
add   $t1, $t1, $s3#eff. Address= 4*i + $s3 is in $t1
lw    $t0, 0($t1)
add   $s1, $s2, $t0
```

Machine Language

- Instructions, like registers and words of data, are also 32 bits long
  - Example: add $t0, $s1, $s2
  - registers have numbers, $t0=9, $s1=17, $s2=18

- Instruction Format:

000000	10001	10010	01001	00000	100000
op	rs	rt	rd	shamt	funct

Machine Language

- Consider the load-word and store-word instructions,
  - What would the regularity principle have us do?
  - New principle: Good design demands a compromise
- Introduce a new type of instruction format
  - I-type for data transfer instructions
  - other format was R-type for register
- Example: lw $t0, 32($s2)

35	18	9	32

op	rs	rt	16-bit number

Register vs. Memory

- Arithmetic instructions operands must be registers,
  - only 32 registers provided
- Compiler associates variables with registers
- What about programs with lots of variables
  - use memory to store variables when all registers are used.
- Memory viewed as a large, single-dimension array, with an address.
- A memory address is an index into the array
- "Byte addressing" means that the index points to a byte of memory.
- For MIPS, a word is 32 bits or 4 bytes.
- 2^{30} words with byte addresses 0, 4, 8, . . . 2^{32} - 4
- 2^{32} bytes with byte addresses from 0 to 2^{32} - 1
- Words are aligned

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

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 $1 \rightarrow 0, 0 \rightarrow 1$

$$x + \overline{x} = 1111...111_2 = -1$$

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

- Example: negate +2
  - $+2 = 0000\ 0000\ \ldots\ 0010_2$
  - $-2 = 1111\ 1111\ \ldots\ 1101_2 + 1$
    $= 1111\ 1111\ \ldots\ 1110_2$

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

Reference:

- **Book:** D. A. Patterson and J. L. Hennessy, **Computer Organization and Design: The Hardware/ Software Interface**, 5th Edition, San Mateo, CA: Morgan and Kaufmann. ISBN: 1-55860-604-1
- **Dr. El-Naga:** ECE-425 Material
- https://www.mips.com/
- https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/mips/index.html
- **Simulator: http://spimsimulator.sourceforge.net/**