# CSE 413 Computer Architecture
# Lecture 3

# Instruction Set

# Introduction

- To command a computer's hardware, you must speak its language.
- The words of a computer's language are called *instructions, and its vocabulary is called an* **instruction set**.
- The chosen instruction set comes from MIPS Technologies, which is an elegant example of the instruction sets designed since the 1980s.
- MIPS (Microprocessor without Interlocked Pipeline Stages) is a reduced instruction set computer (RISC) instruction set architecture (ISA) developed by MIPS Technologies.

# A Basic MIPS Instruction

Every computer must be able to perform arithmetic.

- C code: `a= b + c ;`

- The MIPS assembly language notation

  `add a, b, c`

  instructs a computer to add the two variables b and c and to put their sum in a.

This notation is rigid in that each MIPS arithmetic instruction performs only one operation and must always have exactly three variables.

- Machine code: (hardware-friendly machine instructions)

  `00000010001100100100000000100000`

# Example 1

Translate the following C code into assembly code:

```
a = b + c + d + e;
```

```
add  a, b, c        add  a, b, c
add  a, a, d    or  add  f, d, e
add  a, a, e        add  a, a, f
```

- Instructions are simple: fixed number of operands (unlike C)
- A single line of C code is converted into multiple lines of assembly code
- Some sequences are better than others… the second Sequence needs one more (temporary) variable f

# Example 2

C code  f = (g + h) – (i + j);

Assembly code translation with only add and sub instructions:

```
add  t0, g, h      add  f, g, h
     add  t1,  i, j      or   sub  f, f, i
     sub  f,  t0, t1         sub  f, f, j
```

# Registers

- In C, each "variable" is a location in memory

- Unlike programs in high-level languages, the operands of arithmetic instructions are restricted; they must be from a limited number of special locations built directly in hardware called *registers*.

- To simplify the instructions, we require that each instruction (add, sub) only operate on registers

- One major difference between the variables of a programming language and registers is the limited number of registers, typically 32 on current computers, like MIPS.

# Registers-Cont.

- Each register is 32-bit wide (modern 64-bit architectures have 64-bit wide registers)

- A 32-bit entity (4 bytes) is referred to as a word

Although we could simply write instructions using numbers for registers, from 0 to 31, the MIPS convention is to use two-character names following a dollar sign to represent a register.

- we will use $s0, $s1, . . . for registers that correspond to variables in C and Java programs and $t0, $t1, . . . for temporary registers needed to compile the program into MIPS instructions.

# Example 3

C code  f = (g + h) – (i + j);

The variables f, g, h, i, and j are assigned to the registers $s0, $s1, $s2, $s3, and $s4, respectively.

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

# Memory Operands

- The processor can keep only a small amount of data in registers, but computer memory contains billions of data elements.

Hence, data structures (arrays and structures) are kept in memory.

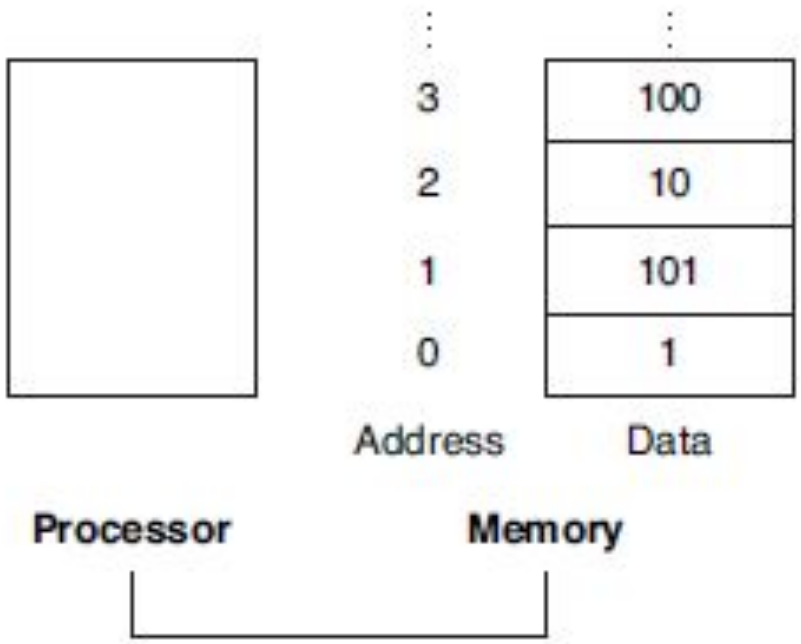
- Arithmetic operations occur only on registers in MIPS instructions; thus, MIPS must include instructions that transfer data between memory and registers.

- Such instructions are called **data transfer instructions**.

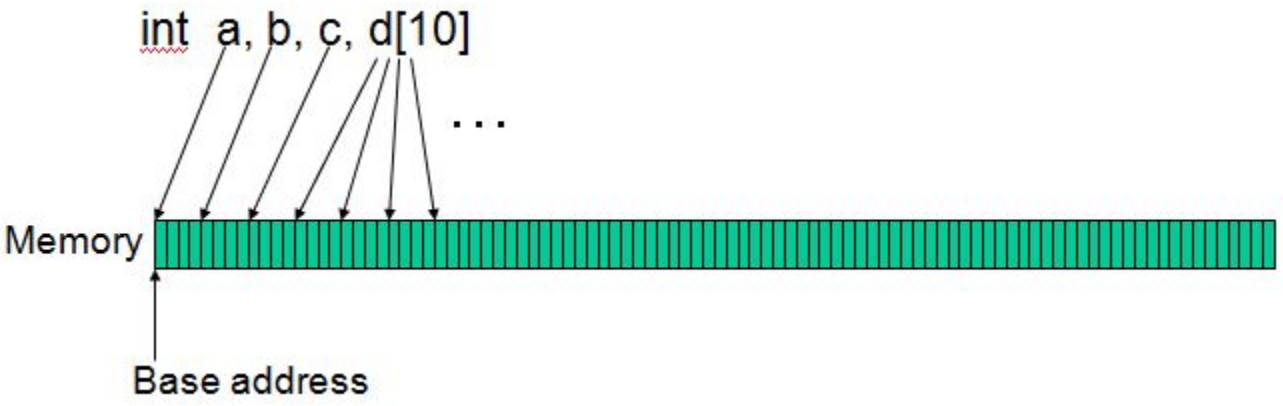
# Memory Operands-Cont.

- To access a word in memory, the instruction must supply the memory **address**. Memory is just a large, single-dimensional array, with the address acting as the index to that array, starting at 0.

- For example, in Figure, the address of the third data element is 2, and the value of Memory[2] is 10.



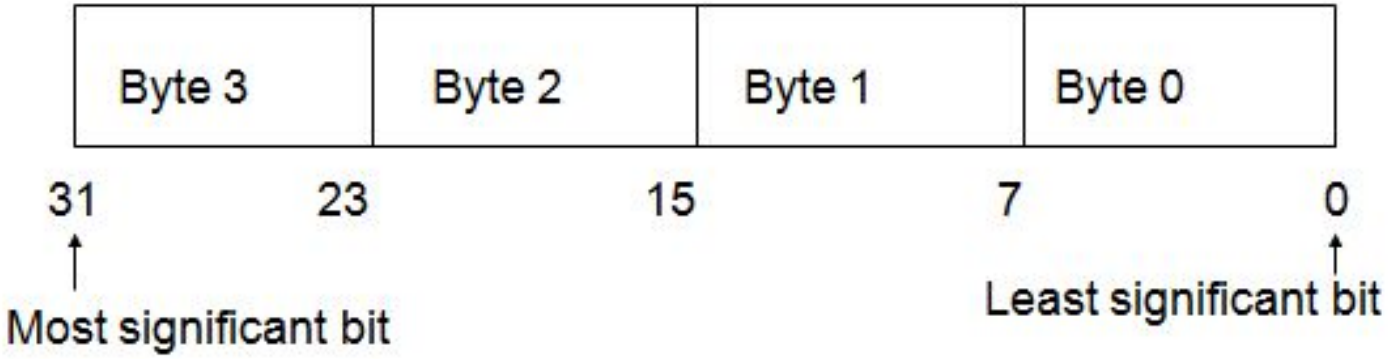
# Example




```
addi   $s0, $zero, 1000      # the program has base address
                                  1000 and this is saved in $s0
                                   $zero is a register that always
                                 equals zero
   addi   $s1, $s0, 0        # this is the address of variable a
   addi   $s2, $s0, 4        # this is the address of variable b
   addi   $s3, $s0, 8        # this is the address of variable c
   addi   $s4, $s0, 12       # this is the address of variable d[10]
```

The **ADDI instruction** performs an addition on both the source register's and stores the result in the destination register.

# Byte Storage



# Big Endian and Little Endian

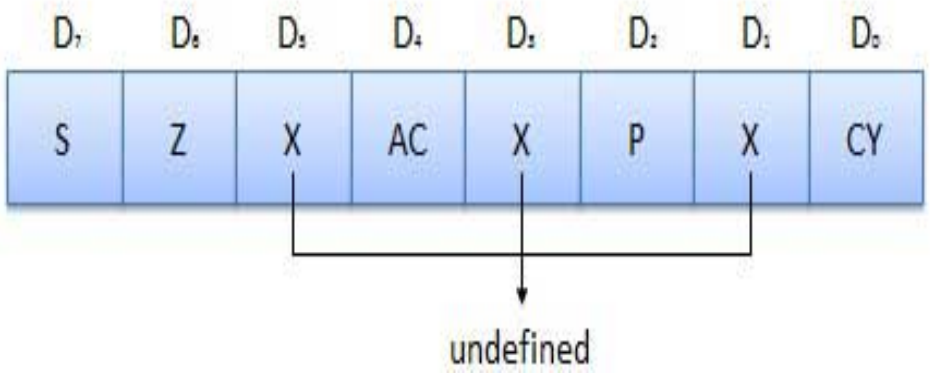
- **Big Endian Byte Order:** The **most significant** byte (the "big end") of the data is placed at the byte with the lowest address. The rest of the data is placed in order in the next three bytes in memory.
- **Little Endian Byte Order:** The **least significant** byte (the "little end") of the data is placed at the byte with the lowest address. The rest of the data is placed in order in the next three bytes in memory.

# ALU

The ALU perform the computing function of microprocessor. It includes the accumulator, temporary register, arithmetic & logic circuit & and five flags. Result is stored in accumulator & flags.

# Accumulator

It is an 8-bit register that is part of ALU. This register is used to store 8-bit data & in performing arithmetic & logic operation. The result of operation is stored in accumulator.



# Flags

In computer science, a flag is a value that acts as a signal for a function or process. The value of the flag is used to determine the next step of a program. Flags are often binary flags, which contain a boolean value (true or false).

**S (Sign) flag** − After the execution of an arithmetic operation, if bit $D_7$ of the result is 1, the sign flag is set. It is used to signed number. In a given byte, if $D_7$ is 1 means negative number. If it is zero means it is a positive number.

**Z (Zero) flag** − The zero flag is set if ALU operation result is 0.

**AC (Auxiliary Carry) flag** − In arithmetic operation, when carry is generated by digit D3 and passed on to digit $D_4$, the AC flag is set. This flag is used only internally BCD operation.

**P (Parity) flag** − After arithmetic or logic operation, if result has even number of 1s, the flag is set. If it has odd number of 1s, flag is reset.

**C (Carry) flag** − If arithmetic operation result is in a carry, the carry flag is set, otherwise it is reset.

# General Purpose Registers

- The four general purpose registers are the AX, BX, CX, and DX registers.
- AX - accumulator, and preferred for most operations.
- BX - base register, typically used to hold the address of a procedure or variable.
- CX - count register, typically used for looping.
- DX - data register, typically used for multiplication and division.

# Word size of memory

A **word** is the unit that a machine uses when working with **memory**. For example, on a 32 bit machine, the **word** is 32 bits long and on a 64 bit is 64 bits long. The **word size** determines the address space.

**Processor**

Microprocessing unit is synonymous to central processing unit, CPU used in traditional computer. Microprocessor (MPU) acts as a device or a group of devices which do the following tasks.

communicate with peripherals devices
provide timing signal
direct data flow
perform computer tasks as specified by the instructions in memory

# Intel 8086 instruction set

The 8086 microprocessor supports 8 types of instructions −

Data Transfer Instructions

Arithmetic Instructions

Bit Manipulation Instructions

String Instructions

Program Execution Transfer Instructions (Branch & Loop Instructions)

Processor Control Instructions

Iteration Control Instructions

Interrupt Instructions

# Data Transfer Instructions

- These instructions are used to transfer the data from the source operand to the destination operand. Following are the list of instructions under this group −

**Instruction to transfer a word**

- **MOV** − Used to copy the byte or word from the provided source to the provided destination.
- **PPUSH** − Used to put a word at the top of the stack.
- **POP** − Used to get a word from the top of the stack to the provided location.
- **PUSHA** − Used to put all the registers into the stack.
- **POPA** − Used to get words from the stack to all registers.
- **XCHG** − Used to exchange the data from two locations.

# Data Transfer Instructions (Contd…)

**Instructions for input and output port transfer**

**IN** − Used to read a byte or word from the provided port to the accumulator.

**OUT** − Used to send out a byte or word from the accumulator to the provided port.

**Instructions to transfer the address**

**LEA** − Used to load the address of operand into the provided register.

**LDS** − Used to load DS register and other provided register from the memory

**LES** − Used to load ES register and other provided register from the memory.

**Instructions to transfer flag registers**

**LAHF** − Used to load AH with the low byte of the flag register.

**SAHF** − Used to store AH register to low byte of the flag register.

**PUSHF** − Used to copy the flag register at the top of the stack.

**POPF** − Used to copy a word at the top of the stack to the flag register.

# Arithmetic Instructions

These instructions are used to perform arithmetic operations like addition, subtraction, multiplication, division, etc.

Following is the list of instructions under this group −

- **Instructions to perform addition**

**ADD** − Used to add the provided byte to byte/word to word.

**ADC** − Used to add with carry.

**INC** − Used to increment the provided byte/word by 1.

**AAA** − Used to adjust ASCII after addition.

**DAA** − Used to adjust the decimal after the addition/subtraction operation.

- **Instructions to perform subtraction**

**SUB** − Used to subtract the byte from byte/word from word.

**SBB** − Used to perform subtraction with borrow.

**DEC** − Used to decrement the provided byte/word by 1.

# Arithmetic Instructions(Contd)

**Instruction to perform multiplication**

**MUL** − Used to multiply unsigned byte by byte/word by word.

**IMUL** − Used to multiply signed byte by byte/word by word.

**AAM** − Used to adjust ASCII codes after multiplication.

**Instructions to perform division**

**DIV** − Used to divide the unsigned word by byte or unsigned double word by word.

**IDIV** − Used to divide the signed word by byte or signed double word by word.

**AAD** − Used to adjust ASCII codes after division.

**CBW** − Used to fill the upper byte of the word with the copies of sign bit of the lower byte.

**CWD** − Used to fill the upper word of the double word with the sign bit of the lower word.

# Bit Manipulation Instructions

These instructions are used to perform operations where data bits are involved, i.e. operations like logical, shift, etc.

Following is the list of instructions under this group −

- **Instructions to perform logical operation**

**NOT** − Used to invert each bit of a byte or word.

**AND** − Used for adding each bit in a byte/word with the corresponding bit in another byte/word.

**OR** − Used to multiply each bit in a byte/word with the corresponding bit in another byte/word.

**XOR** − Used to perform Exclusive-OR operation over each bit in a byte/word with the corresponding bit in another byte/word.

**TEST** − Used to add operands to update flags, without affecting operands.

# Bit Manipulation Instructions

- **Instructions to perform shift operations**

**SHL/SAL** − Used to shift bits of a byte/word towards left and put zero(S) in LSBs.

**SHR** − Used to shift bits of a byte/word towards the right and put zero(S) in MSBs.

**SAR** − Used to shift bits of a byte/word towards the right and copy the old MSB into the new MSB.

- **Instructions to perform rotate operations**

**ROL** − Used to rotate bits of byte/word towards the left, i.e. MSB to LSB and to Carry Flag [CF].

**ROR** − Used to rotate bits of byte/word towards the right, i.e. LSB to MSB and to Carry Flag [CF].

**RCR** − Used to rotate bits of byte/word towards the right, i.e. LSB to CF and CF to MSB.

**RCL** − Used to rotate bits of byte/word towards the left, i.e. MSB to CF and CF to LSB.

# String Instructions

•String is a group of bytes/words and their memory is always allocated in a sequential order.

•Following is the list of instructions under this group −

**REP** − Used to repeat the given instruction till CX ≠ 0.

**REPE/REPZ** − Used to repeat the given instruction until CX = 0 or zero flag ZF = 1.

**REPNE/REPNZ** − Used to repeat the given instruction until CX = 0 or zero flag ZF = 1.

**MOVS/MOVSB/MOVSW** − Used to move the byte/word from one string to another.

**COMS/COMPSB/COMPSW** − Used to compare two string bytes/words.

**INS/INSB/INSW** − Used as an input string/byte/word from the I/O port to the provided memory location.

**OUTS/OUTSB/OUTSW** − Used as an output string/byte/word from the provided memory location to the I/O port.

# Program Execution Transfer Instructions (Branch and Loop Instructions)

These instructions are used to transfer/branch the instructions during an execution. It includes the following instructions −

Instructions to transfer the instruction during an execution without any condition −

**CALL** − Used to call a procedure and save their return address to the stack.

**RET** − Used to return from the procedure to the main program.

**JMP** − Used to jump to the provided address to proceed to the next instruction.

- Instructions to transfer the instruction during an execution with some conditions −

**JA/JNBE** − Used to jump if above/not below/equal instruction satisfies.

**JAE/JNB** − Used to jump if above/not below instruction satisfies.

**JBE/JNA** − Used to jump if below/equal/ not above instruction satisfies.

**JC** − Used to jump if carry flag CF = 1

**JE/JZ** − Used to jump if equal/zero flag ZF = 1

# Processor Control Instructions

These instructions are used to control the processor action by setting/resetting the flag values.

Following are the instructions under this group −

**STC** − Used to set carry flag CF to 1

**CLC** − Used to clear/reset carry flag CF to 0

**CMC** − Used to put complement at the state of carry flag CF.

**STD** − Used to set the direction flag DF to 1

**CLD** − Used to clear/reset the direction flag DF to 0

**STI** − Used to set the interrupt enable flag to 1, i.e., enable INTR input.

**CLI** − Used to clear the interrupt enable flag to 0, i.e., disable INTR input.

# Iteration Control Instructions

These instructions are used to execute the given instructions for number of times. Following is the list of instructions under this group −

**LOOP** − Used to loop a group of instructions until the condition satisfies, i.e., CX = 0

**LOOPE/LOOPZ** − Used to loop a group of instructions till it satisfies ZF = 1 & CX = 0

**LOOPNE/LOOPNZ** − Used to loop a group of instructions till it satisfies ZF = 0 & CX = 0

**JCXZ** − Used to jump to the provided address if CX = 0

# Interrupt Instructions

These instructions are used to call the interrupt during program execution.

**INT** − Used to interrupt the program during execution and calling service specified.
**INTO** − Used to interrupt the program during execution if OF = 1
**IRET** − Used to return from interrupt service to the main program

Thank you

☺