Outline of Lecture

1. Introduction to Instruction Set Architecture
2. Operations of the Computer Hardware
3. Operands of the Computer Hardware

Instruction Set Architecture

- In order to use the hardware of a computer, we must *speak* its language - It is the portion of the computer which is visible to the programmer and the compiler writer.
- The words of a machine (computer) language are called *instructions*, and its vocabulary is called an *instruction set*.
- We will see the instruction set of a real computer, both in the form written by the programmer and the form read by the machine.

The MIPS

The real computer we are using is called MIPS and comes from the MIPS computer company. MIPS, was later sold to SGI, and all SGI machines are based on the original MIPS architecture.

- We will also analyze ***why*** these computer architects came up with such an instruction set for MIPS.

Operations of the Computer Hardware

- Every computer must be able to perform ***arithmetic*** operations:

```
add  a,  b,  c
```

It is a MIPS ***assembly language*** instructions that Instructs a computer to add two variables b and c and put their sum into a.

Sample Program

To put the sum of 4 variables b, c, d, and e into variable a, we need the following sequence of operations:

```
add a, b, c        # The sum of b and c is
                     placed in a

add a, a, d        # The sum of b, c and d is
                     now in a

add a, a, e        # The sum of b, c, d and e
                     is now in a.
```

➜ **The words to the right of each instruction are *comments*.**

➜ **Each line of code can contain at most *one* instruction.**

➜ **The number of *operands* in each instruction is exactly three.**

Instruction Format

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add a, b, c`	a = b + c	Always 3 operands
	subtract	`sub a, b, c`	a = b -c	Always 3 operands

Why do we insist that we always use 3 operands (why not allow 4, 5, etc. operands)?

The hardware for a variable number of operands (e.g., pentium) is more complicated than the hardware for a fixed number of operands.

Principle #1: **Simplicity is better than complexity in hardware design.**

Example

Translate the following C statement into a MIPS assembly language instructions (this is the job of the *compiler*):

```
f = (g + h) - (i + j);
```

Answer

```
add t0, g, h   # temporary variable t0 contains
                 g+h

add t1, i, j   # temporary variable t1 contains
                 i+j

sub f, t0, t1 # f gets t0-t1, or (g+h) - (i+j)
```

Operands of the Computer Hardware

- Unlike programs in high-level languages, the operands of assembly language instructions cannot be any variables - they must be from a limited number of locations called ***registers***.
- Registers are fast temporary memory locations inside the processor - they are visible to the programmer that can be used to hold variables.
- The size of a register in a MIPS computer is ***32 bits*** - groups of 32 bits are given the name ***word*** in the MIPS architecture.
- The MIPS computer has ***32 registers***, using the notation `$0, $1, ..., $31` to represent them.

32 registers may not be enough to hold the potential large number of variables in big programs.

However, a large number of registers complicates the design of the processor and increases its clock cycle.

A computer designer should strike a ***balance*** between providing a large number of registers and a fast processor.

Principle #2: **Having a small number of registers (e.g., 16-128) leads to a faster design of the processor.**

Example

Given the following C statement:

```
f = (g + h) - (i + j);
```

Assume the compiler associates the variables `f`, `g`, `h`, `i`, and `j` to the registers `$s0`, `$s1`, `$s2`, `$s3`, and `$s4`, respectively

What is the compiled MIPS assembly language code?

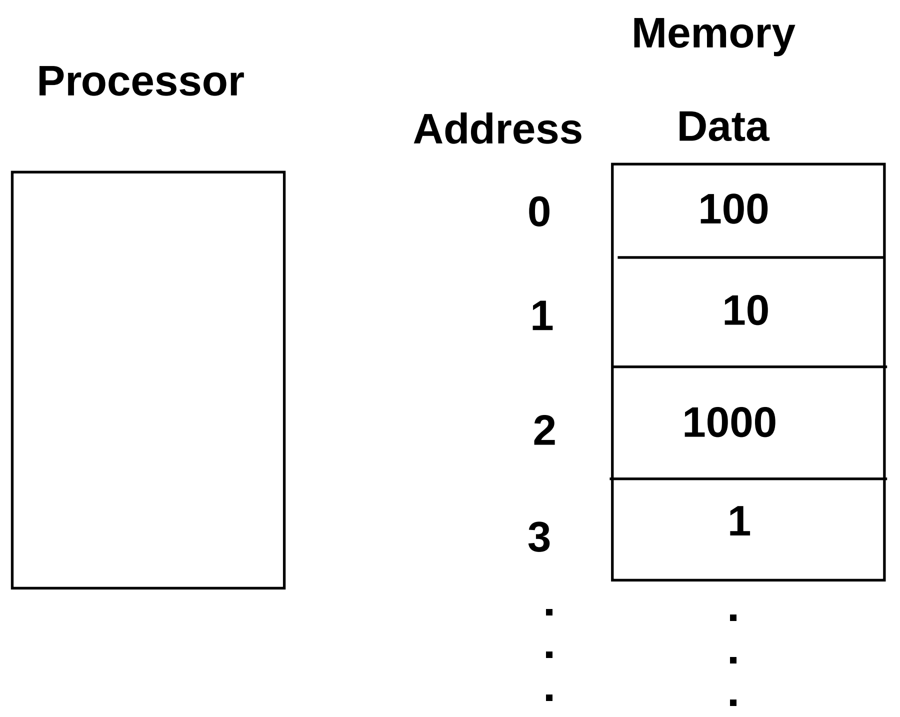
Answer

```
add $t0, $s1, $s2 # Register $t0 contains g+h

add $t1, $s3, $s4 # Register $t1 contains i+j

sub $s0, $t0, $t1 # f gets $t0-$t1, or (g+h) -
                      (i+j)
```

- What if we have a program that manipulates a large array of numbers - they cannot *all* be stored in the registers of the MIPS processor.
- In this case, the elements of the array would be stored in the ***memory*** of the MIPS computer.
- The memory is a large storage space that can store millions of data elements.
- When we need to perform an operation on certain elements of this array, we ***transfer*** these elements from the memory to the registers - MIPS cannot perform operations directly on data elements stored in memory (certain computers can).
- These instructions are called ***data transfer*** instructions.

What is an Address

- To access a word in memory, the data transfer instruction must supply its ***address*** (Memory[2] = 1000 in the example below).



- The data transfer instruction that moves data from memory to a register is called ***`load`***.

- The MIPS assembly language notation for this data transfer instruction is `lw` which stands for `load word`.

- The format of `lw` is such that:

  1) It should contain the ***start address*** of the array,

  2) It should contain a register that contains the ***index*** of the element of the array to be loaded

(e.g., `lw $27, Abegin($12)`).

Address of array element = `Abegin` + ***content*** of register `$12`.

- Register `$12`, in the above example, is called `index register`.

Example

Assume that A is an array of 100 elements and that the compiler has associated the variables *g*, *h* with registers $s1, and $s2. Let us assume also that the starting address, also called base address, is in $s3.

Translate this C statements into MIPS assembly code:

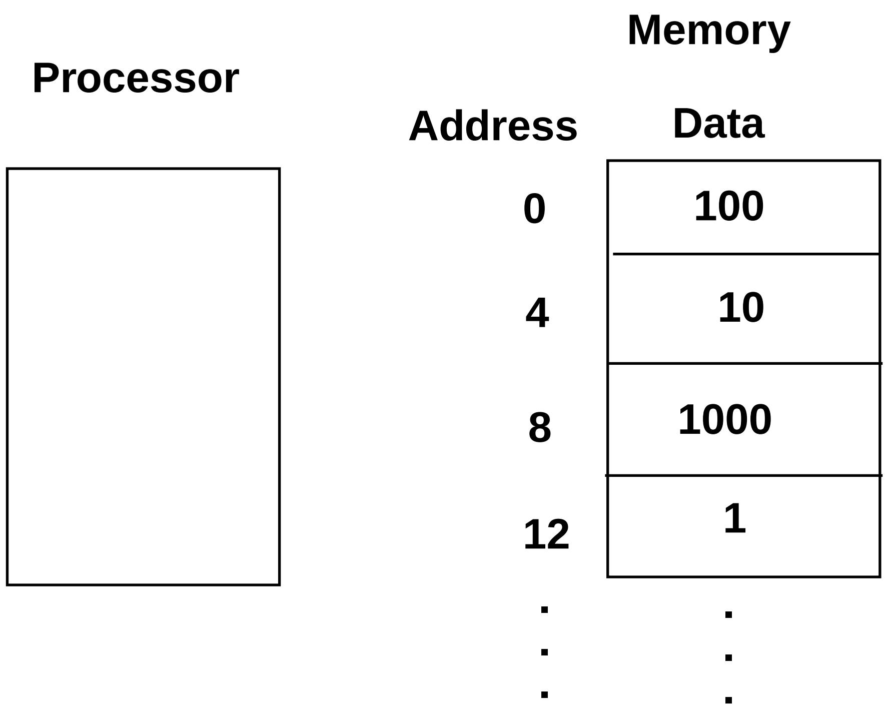
```
g = h + A[8]
```

Answer

```
lw $t0, 8($s3) # Temporary reg $t0 gets A[8]
add $s1, $s2, $t0    # g = h + A[i]
```

We have shown how to address words (32 bits) in MIPS. However, most computers - including MIPS - addresses individual ***bytes*** (8 bits) as well.

As a result, real memory addresses are as follows:



- The MIPS assembly language instruction which is complimentary to `load` is called `store`.
- The `store` instruction transfers data from a register to a memory location.
- The MIPS assembly language notation for this data transfer instruction is `sw` which stands for `store word`.
- The format of `sw` is such that (similar to `lw`):

  1) It should contain the ***start address*** of the array,

  2) It should contain a register that contains the ***index*** of the element of the array to be stored

Example

Assume the variable h is associated with the register $s2. and the base address of the array A is in $s3.

What is the MIPS assembly language code for the C statement below:

```
A[12] = h + A[8]
```

Answer

```
lw $t0, 32($s3)  # Temporary reg $t0 gets A[8]
add $t0, $s2, $t0 # Temporary reg $t0 gets
                                  h+A[8]
sw $t0, 48($s3) # Stores h+A[8] back into A[12]
```

Summary of MIPS Assembly language

The instructions seen so far:

Name	Example	Comments
32 registers	`$s0, $s1, .. $t0, $t1,..`	Fast storage locations for data. In MIPS, data must be stored in registers to perform arithmetic.
2^{30} memory words	`Memory[0], Memory[4], ..., Memory [429967292]`	Access only by data transfer instructions in MIPS. MIPS use byte addresses, so sequential words differ by 4.

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`	3 operands; data in registers
	subtract	`sub $s1, $s2, $s3`	`$s1 = $s2 - $s3`	3 operands; data in registers
Data transfer	load word	`lw $1, 100($2)`	`$s1 = Memory [$s2+100]`	Data from memory to register
	store word	`sw $1, 100($2)`	`Memory [$s2+100] = $s1`	Data from register to memory