CSE 2105
COMPUTER ARCHITECTURE

Md. Sabab Zulfiker,
Lecturer,
Department of CSE,
BSMRU, Kishoreganj.

Books

- Computer Organization and Design- The Hardware/Software Interface **By** David A. Patterson & John L. Hennessy
- Computer Organization and Design- The Hardware/Software Interface: RISC-V Edition **By** David A. Patterson & John L. Hennessy
- Structured Computer Organization **By** Andrew S. Tanenbaum

Instructions: Language of the Computer-I

Instructions and Instruction Set

- The words of a computer's language are called **instructions**, and its vocabulary is called an **instruction set**.
- The vocabulary of commands understood by a given architecture is called **instruction set**.
- You might think that the languages of computers would be as diverse as those of people, but in reality computer languages are quite similar, more like regional dialects than like independent languages. Hence, once you learn one, it is easy to pick up others.
- Computer designers have a common goal: to find a language that makes it easy to build the hardware and the compiler while maximizing performance and minimizing cost and energy.

Stored-program concept

- **Stored-program concept:** The idea that instructions and data of many types can be stored in memory as numbers, leading to the stored program computer.

Operations of the Computer Hardware

- Every computer must be able to perform arithmetic. 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.

- MIPS arithmetic instruction performs only one operation and must always have exactly three variables.
- For example, suppose we want to place the sum of four variables **`b, c, d,`** and **`e`** into variable **`a`**.
- The following sequence of instructions adds the four variables:

Operations of the Computer Hardware

```
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
```

- Thus, it takes three instructions to sum the four variables. The words to the right of the sharp symbol (#) on each line above are comments for the human reader, so the computer ignores them.
- Each line of this language can contain at most one instruction.
- Another difference from C is that comments always terminate at the end of a line.

Compiling Two C Assignment Statements into MIPS

This segment of a C program contains the five variables `a, b, c, d,` and `e`. Since Java evolved from C, this example and the next few work for either high-level programming language:

```
a = b + c;

d = a – e;
```

The translation from C to MIPS assembly language instructions is performed by the compiler. **Show the MIPS code produced by a compiler.**

Compiling Two C Assignment Statements into MIPS

Solution:

A MIPS instruction operates on two source operands and places the result in one destination operand. Hence, the two simple statements above compile directly into these two MIPS assembly language instructions:

```
add a, b, c
sub d, a, e
```

Compiling a Complex C Assignment into MIPS

A somewhat complex statement contains the five variables **`f, g, h, i,`** and **`j`**:

```
f = (g + h) – (i + j);
```

What might a C compiler produce?

Solution:

The compiler must break this statement into several assembly instructions, since only one operation is performed per MIPS instruction. The first MIPS instruction calculates the sum of **`g`** and **`h`**. We must place the result somewhere, so the compiler creates a temporary variable, called **`t0`**:

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

Compiling a Complex C Assignment into MIPS

Although the next operation is subtract, we need to calculate the sum of **`i`** and **`j`** before we can subtract. Thus, the second instruction places the sum of **`i`** and **`j`** in another temporary variable created by the compiler, called **`t1`**:

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

Finally, the subtract instruction subtracts the second sum from the first and places the difference in the variable **`f`**, completing the compiled code:

```
sub f,t0,t1 #f gets t0 – t1, which is (g + h) – (i + j)
```

MIPS Registers

- MIPS has 32 general-purpose registers and another 32 floating-point registers. Registers all begin with a dollar-symbol ($). The floating point registers are named $f0, $f1, ..., $f31. The general-purpose registers have both names and numbers, and are listed below.

Number	Name	Comments
$0	$zero, $r0	Always zero
$1	$at	Reserved for assembler
$2, $3	$v0, $v1	First and second return values, respectively
$4, ..., $7	$a0, ..., $a3	First four arguments to functions
$8, ..., $15	$t0, ..., $t7	Temporary registers
$16, ..., $23	$s0, ..., $s7	Saved registers
$24, $25	$t8, $t9	More temporary registers
$26, $27	$k0, $k1	Reserved for kernel (operating system)
$28	$gp	Global pointer
$29	$sp	Stack pointer
$30	$fp	Frame pointer
$31	$ra	Return address

MIPS Registers

- The size of a register in the MIPS architecture is 32 bits; groups of 32 bits occur so frequently that they are given the name word in the MIPS architecture.
- **Word:** The natural unit of access in a computer, usually a group of 32 bits; corresponds to the size of a register in the MIPS architecture.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three register operands
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 – $s3	Three register operands
	add immediate	`addi $s1,$s2,20`	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	`lw $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	`sw $s1,20($s2)`	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	`lh $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	`lhu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	`sh $s1,20($s2)`	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	`lb $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	`lbu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	`sb $s1,20($s2)`	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	`ll $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	`sc $s1,20($s2)`	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	`lui $s1,20`	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Logical	and	and $s1,$s2,$s3	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	or $s1,$s2,$s3	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	nor $s1,$s2,$s3	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	andi $s1,$s2,20	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	ori $s1,$s2,20	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl $s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
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 on less than unsigned	`sltu $s1,$s2,$s3`	if (`$s2` < `$s3`) `$s1` = 1; else `$s1` = 0	Compare less than unsigned
	set less than immediate	`slti $s1,$s2,20`	if (`$s2` < 20) `$s1` = 1; else `$s1` = 0	Compare less than constant
	set less than immediate unsigned	`sltiu $s1,$s2,20`	if (`$s2` < 20) `$s1` = 1; else `$s1` = 0	Compare less than constant unsigned
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

Compiling a C Assignment Using Registers

It is the compiler's job to associate program variables with registers. Take, for instance, the assignment statement from our earlier example:

```
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. What is the compiled MIPS code?

Compiling a C Assignment Using Registers

Solution:

The compiled program is very similar to the prior example, except we replace the variables with the register names mentioned above plus two temporary registers, **`$t0`** and **`$t1`**, which correspond to the temporary variables above:

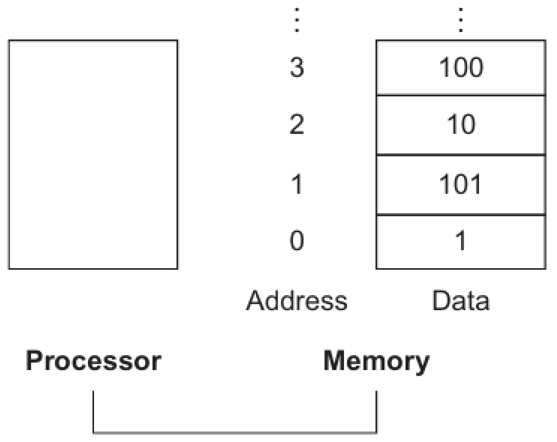
```
add $t0,$s1,$s2 #register $t0 contains g + h

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

sub $s0,$t0,$t1 #f gets $t0 – $t1, which is (g + h)–(i + j)
```

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**.
- To access a word in memory, the instruction must supply the memory address. **Address** is a value used to delineate the location of a specific data element within a memory array.

Memory Operands

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



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

Memory Operands

- The format of the load instruction is the name of the operation followed by the register to be loaded, then a constant and register used to access memory.
- The sum of the constant portion of the instruction and the contents of the second register forms the memory address.
- The actual MIPS name for this instruction is **`lw`**, standing for load word.

Compiling an Assignment When an Operand Is in Memory

Let's assume that **A** is an array of 100 words and that the compiler has associated the variables **g** and **h** with the registers **$s1** and **$s2** as before. Let's also assume that the starting address, or base address, of the array is in **$s3**. Compile this C assignment statement:

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

Solution:

We must first transfer **A[8]** to a register. The address of this array element is the sum of the base of the array **A**, found in register **$s3**, plus the number to select element **8**. The data should be placed in a temporary register for use in the next instruction.

```
lw $t0,8($s3) #Temporary reg $t0 gets A[8]
```

Compiling an Assignment When an Operand Is in Memory

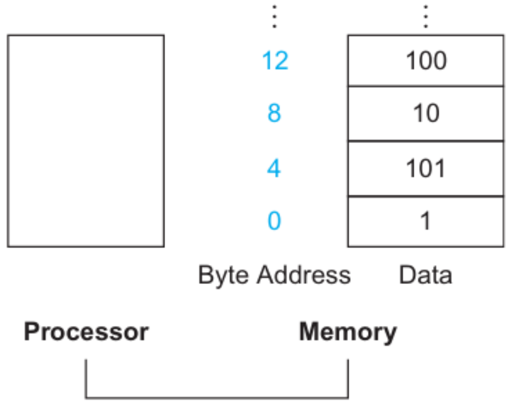
The instruction must add **`h`** (contained in **`$s2`**) to **`A[8]`** (contained in **`$t0`**) and put the sum in the register corresponding to **`g`** (associated with **`$s1`**):

```
add   $s1,$s2,$t0 #g = h + A[8]
```

The constant in a data transfer instruction **`(8)`** is called the **off set**, and the register added to form the address **`($s3)`** is called the **base register**.

Byte Addressing

- The address of a word matches the address of one of the 4 bytes within the word, and addresses of sequential words differ by 4.
- For example, the following Figure shows the actual MIPS addresses for the words; the byte address of the third word is 8.



- In MIPS, words must start at addresses that are multiples of 4. This requirement is called an **alignment restriction**.

Byte Addressing

- Byte addressing also affects the array index. To get the proper byte address in the code above, the off set to be added to the base register $s3 must be 4×8, or 32.
- The instruction complementary to load is traditionally called store; it copies data from a register to memory.
- The format of a store is similar to that of a load: the name of the operation, followed by the register to be stored, then off set to select the array element, and finally the base register.
- The actual MIPS name is `sw`, standing for store word.

Compiling Using Load and Store

Assume variable **`h`** is associated with register **`$s2`** and the base address of the array **`A`** is in **`$s3`**. What is the MIPS assembly code for the C assignment statement below?

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

Solution:

The first two instructions are the same as in the prior example, except this time we use the proper off set for byte addressing in the load word instruction to select **`A[8]`**, and the add instruction places the sum in **`$t0`**:

```
lw $t0,32($s3) #Temporary reg $t0 gets A[8]
add $t0,$s2,$t0 #Temporary reg $t0 gets h + A[8]
```

Compiling Using Load and Store

The final instruction stores the sum into **A[12],** using 48 (4×12) as the offset and register **$s3** as the base register.

```
sw $t0,48($s3) #Stores h + A[8] back into A[12]
```

Constant or Immediate Operands

- For example, to add the constant 4 to register **$s3**, we could use the code:

```
lw $t0, AddrConstant4($s1)    #$t0 = constant 4
add $s3,$s3,$t0               #$s3 = $s3 + $t0 ($t0 == 4)
```

Assume that **$s1+AddrConstant4** is the memory address of the constant 4.

- An alternative that avoids the load instruction is to offer versions of the arithmetic instructions in which one operand is a constant.
- This quick add instruction with one constant operand is called **add immediate** or **addi**.
- To add the constant 4 to register **$s3**, we just write:

```
addi $s3,$s3,4             # $s3 = $s3 + 4
```

Thank You