COMP2611: Computer Organization

Instructions: Language of the Computer

Major Goal

To command a computer's hardware, you must speak its language

- To learn a subset of the **MIPS assembly language**
  - The "words" of a machine language are called **instructions**
  - Its "vocabulary" is called an **instruction set**.
  - In the form written by the programmer → **assembly language**
  - In the form the computer can understand → **machine language**
- To learn the design principles for **instruction set architecture (ISA)**

What is MIPS?

MIPS (<u>M</u>icroprocessor without <u>I</u>nterlocked <u>P</u>ipeline <u>S</u>tages)

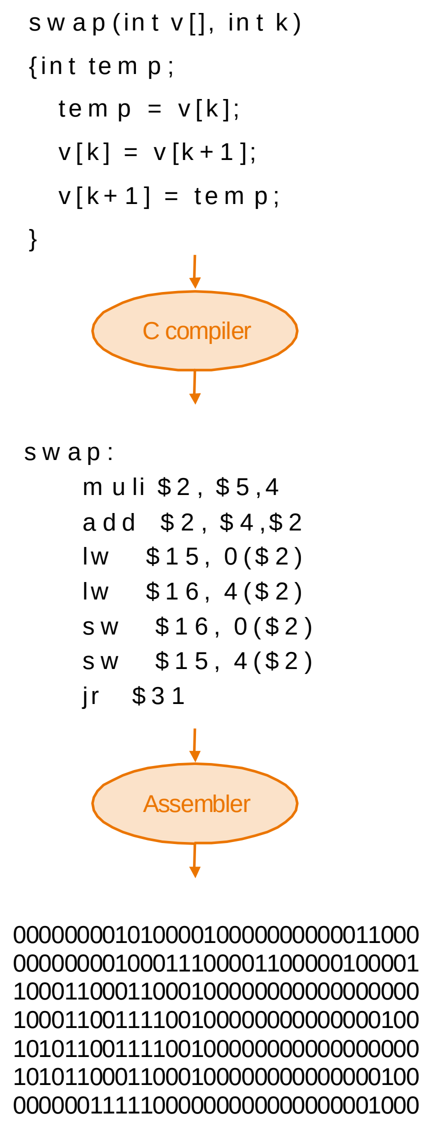
- A widely used microprocessor architecture
- e.g. Silicon Graphic (SGI), ATI Technologies, Broadcom, Cisco, NEC, Nintendo, Sony PlayStation, Texas Instruments (TI), Toshiba, embedded systems, Windows CE devices

What is ISA

- An instruction set, or instruction set architecture (ISA), is the part of the processor that is visible to the programmer or compiler writer. It includes
  - the native data types,
  - instructions,
  - registers,
  - addressing modes,
  - memory architecture,
  - interrupt and exception handling,
  - and external I/O.
- An ISA includes a specification of the set of opcodes (machine language), and the native commands implemented by a particular processor.

ISA vs. Assembly Language

- ISA vs. Assembly Language
  - "ISA" standardizes a public interface to a processor that should be used as the basis to write programs
  - Assembly Language is a term for a programming language.
  - Ideally for each ISA there is an Assembly Language,
  - But it is not all that uncommon that more than one or subtle variations can exist between Assembly Languages for a specific ISA.
  - The Assembly Language is essentially defined by the Assembler



```
swap(int v[], int k)
{int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

```
swap:
      muli $2, $5,4
      add  $2, $4,$2
      lw   $15, 0($2)
      lw   $16, 4($2)
      sw   $16, 0($2)
      sw   $15, 4($2)
      jr   $31
```

```
00000000101000010000000000011000
00000000100011100001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

Why ISA?

Imagine that Intel processors do not have an ISA

- It means the vocabulary of hardware can change from time to time
- No guarantee of how the instructions will look like in the next product

- From programmers' point of view
  - Potentially **need to re-program** every time we upgrade our computer. It would be a nightmare!
- From system designers' point of view:
  - Hardware improvement (for performance, or power efficiency, etc) may lead to **incompatibility** with existing applications

With ISA abstraction, all these problem are resolved

- All software developers have to do is conforming to machine's ISA
  - No need to worry about how hardware implements the instructions
- All system designers have to do is making sure new processor implementation backward compatible to ISA's definition, instead of existing applications

Therefore, it is critical to have a good ISA design!

Design Principles for ISA

- Simplicity favors regularity
- Smaller is faster
- Make common case fast
- Good design demands good compromises

1. Basic MIPS Instructions

Operations of Computer Hardware

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

- Example of a MIPS arithmetic instruction

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

  - It is equivalent to a = b + c in C++

```
sub x, y, z
```

  - It is equivalent to x = y - z in C++

Operations of Computer Hardware (cont'd)

- Another example (adding four variables: a = b + c + d + e)

```
add a, b, c    # sum of b & c is placed in a
add a, a, d    # sum of b, c & d is now in a
add a, a, e    # sum of b, c, d & e is now in a
```

  - Three instructions are needed! Why?
    - Because each `add` instruction can only have three variables (called **operands**) in MIPS architecture

Why a fixed number of operands?

MIPS Instruction Format

Design Principle #1

Simplicity favors regularity

- Each instruction has a fixed number of operands in MIPS
- Intel architecture supports a variable number of operands

- Why fixed number instead of variable number of operands?
  - ★ The hardware is less complicated for a fixed number of operands

More Examples

- Consider a high-level language statement with 5 vars (`f`, `g`, `h`, `i`, `j`)

```
f = (g + h) – (i + j);
```

- Translated to MIPS instructions (to be modified to a more realistic solution later):

```
add t0, g, h    # temp variable t0 contains g+h
add t1, i, j    # temp variable t1 contains i+j
sub f, t0, t1   # f gets t0 – t1, or (g+h) –
                # (i+j)
```

Notes:

- Each line of code contain at most **one** instruction
- Words to the right of **#** symbol are **comments** for the human reader
- **Comments** are entirely ignored by the computer

Register Operand

Unlike programs in high-level languages, the operands of arithmetic instructions cannot be any **variables**. They must be from a limited number of special locations called **registers**.

What are registers?

- ❑ Fast temporary storage inside the processor used to hold variables.
- ❑ Size of a register in the MIPS architecture is 32 bits.
  - ❍ Each group of 32 bits is called a word in the MIPS architecture.
- ❑ MIPS architecture has 32 registers

Variable vs. Register

- ❑ Variable is a logical storage, # of variables can be unlimited
- ❑ Register is a physical storage, # of registers is limited

What happens if not enough registers to hold all the variables?

Different Types of Registers in MIPS

MIPS has 32 general purpose registers, each is of 32 bits in length

- Registers that correspond to variables in a high-level program are denoted as `$s0`, `$s1`, … , `$s7`
- Temporary registers needed to compile the program into MIPS instructions are denoted as `$t0`, `$t1`, … , `$t7`
- $zero, a special register holding a constant value 0, read-only (i.e. not modifiable), (will be explained why we need it)
- Others will be introduced much later

Completing the Previous Example

- Translate the following statement to MIPS assembly language

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

- Using registers `$s0`, `$s1`, `$s2`, `$s3`, `$s4`, to hold variables f, g, h, i, j

- **Answer**:

```
add $t0, $s1, $s2     # reg $t0 contains g+h
add $t1, $s3, $s4     # reg $t1 contains i+j
sub $s0, $t0, $t1     # f gets $t0 - $t1
```

How Many Registers Do We Need?

If the number of registers is

- **Too few**:
  - not enough to hold large number of variables in a program
- **Too many**:
  - more complicated processor design
  - increased clock cycle time ⇨ obstacle to improving performance

The computer architect should strike a good balance between providing a large number of registers and keeping the clock cycle short.

How Many Registers Do We Need? (cont'd)

Design Principle #2

Smaller is faster

- Having a small enough number of registers leads to a faster processor
- Why?
  - ★ Larger number of registers, longer electronic signals must travel

Data Transfer in MIPS

What if a program manipulates a large number of elements?

- Not possible to store elements all at once in registers inside processor
- Ex: Large composite data like arrays, structures and dynamic data are kept in the memory
    - memory provides large storage for millions of data elements.

MIPS' design disallows values stored in memory to be manipulated directly

Then, how does MIPS use such data?

- Data must be transferred from memory to a register before manipulation and the results are stored back to memory
- We need Data transfer instructions
    - **load** moves data from memory to a register, e.g. **lw** (load a word)
    - **store** moves data from a register to memory, e.g. **sw** (store a word)

Example with Data Transfer

- Translate the following statement to MIPS assembly language

```
f = g + h;
```

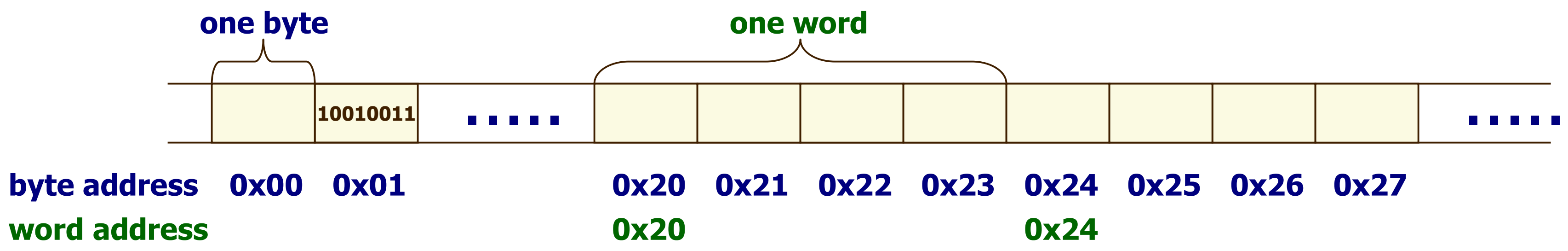
- **Answer**:

```
lw   $s1, g            # load variable into register
lw   $s2, h            # load variable into register
add  $t0, $s1, $s2     # reg $t0 contains g+h
sw   $t0, f            # store (g+h) to f
```

Next, we need to express g, h, and f in terms of memory location!

Addressing Memory Location

- **Memory** is a consecutive arrangement of storage locations.



- Each memory location is indexed by an **address**.
  - Most architectures address individual bytes
  - Addresses of sequential 8-bit bytes differ by 1
  - Addresses of sequential 32-bit words differ by 4

- To specify the address of the memory location of any array element in assembly language, we need two parts:
  - **Base address**: starting address of an array
  - **Offset**: distance of target location from starting address
    it is a constant that can be either positive or negative

Memory Operand

`A` is an array of 100 words

- How can we perform `A[12] = h + A[8];` in MIPS?

Assume `$s0` store the starting location of array `A` (i.e., address of A[0])

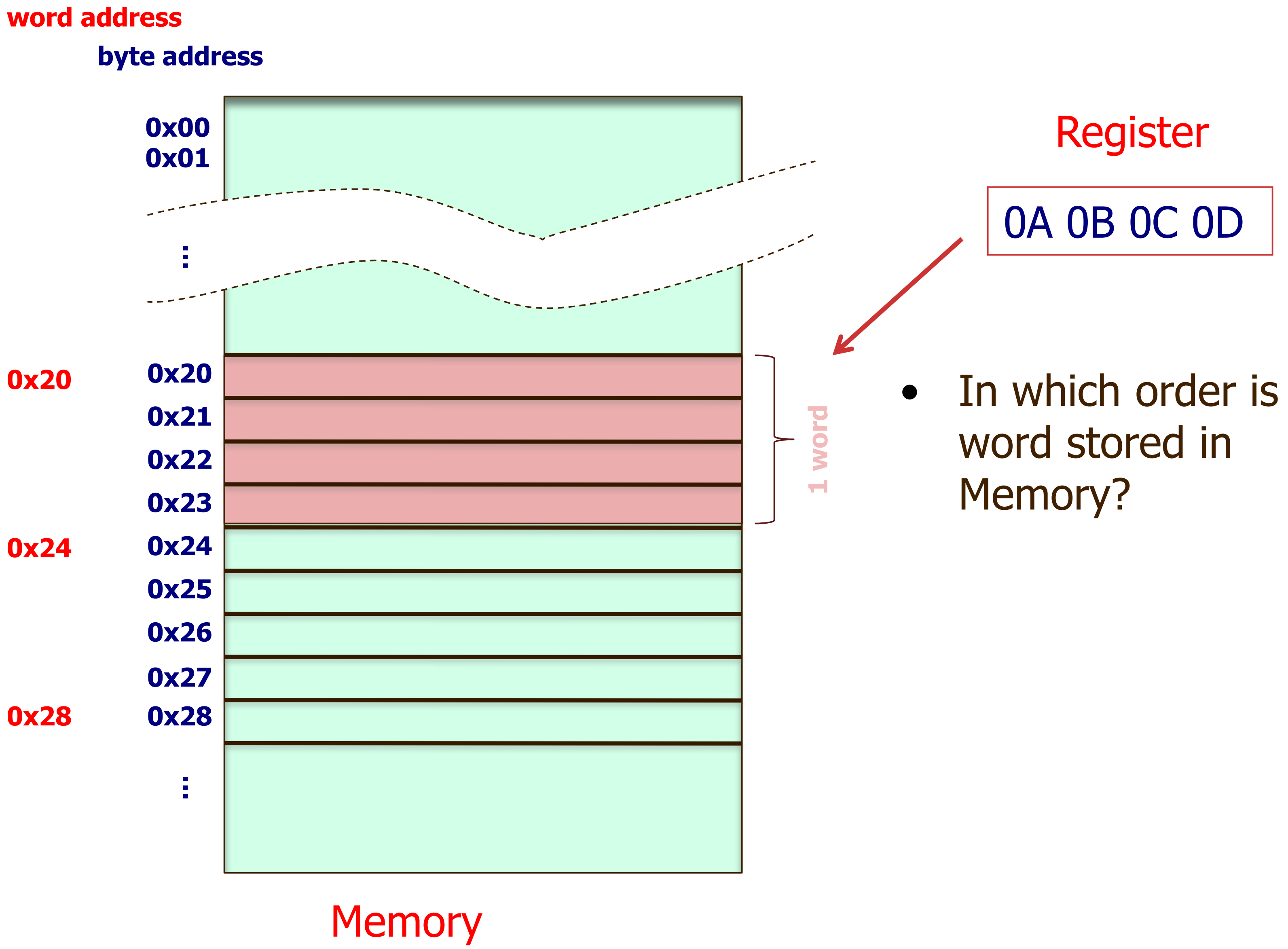
Assume `$s1` store the value of `h`

- **Answer**:

```
# temp reg $t0 gets A[8]
lw  $t0, 32($s0)          # address of A[8] : $s0 + 32
                   ?
# $t0 = h + A[8]
add $t0, $s1, $t0
# stores h + A[8] to A[12]
sw  $t0, 48($s0)
```

- **Remark**: `$s0` is used as a **base register**, "32" and "48" are **offsets**

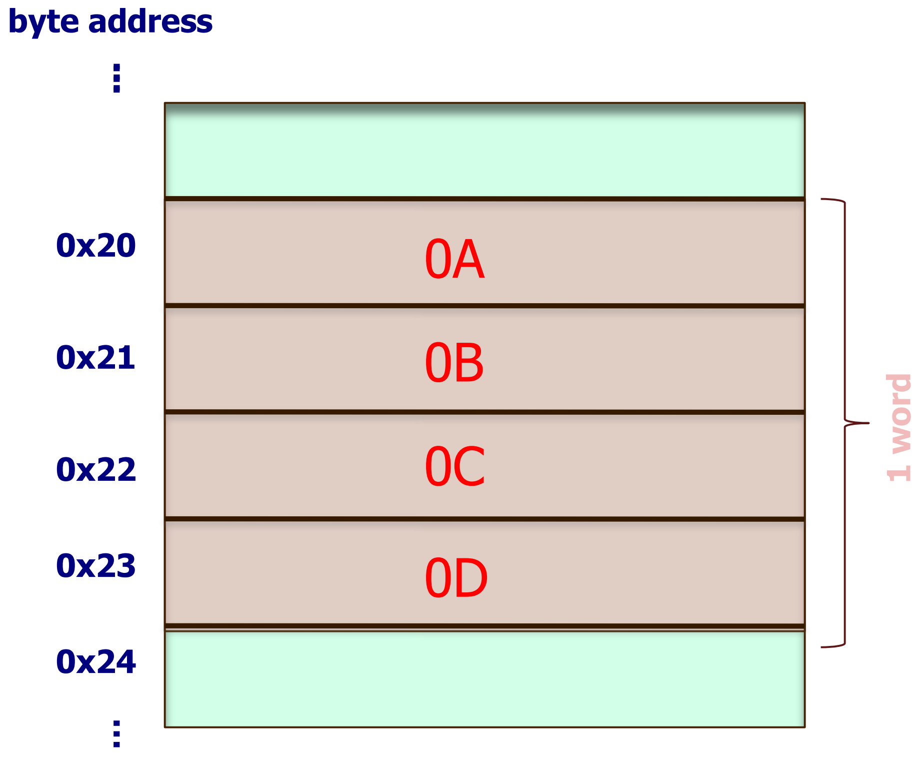
Endianness (byte order): Little or Big?

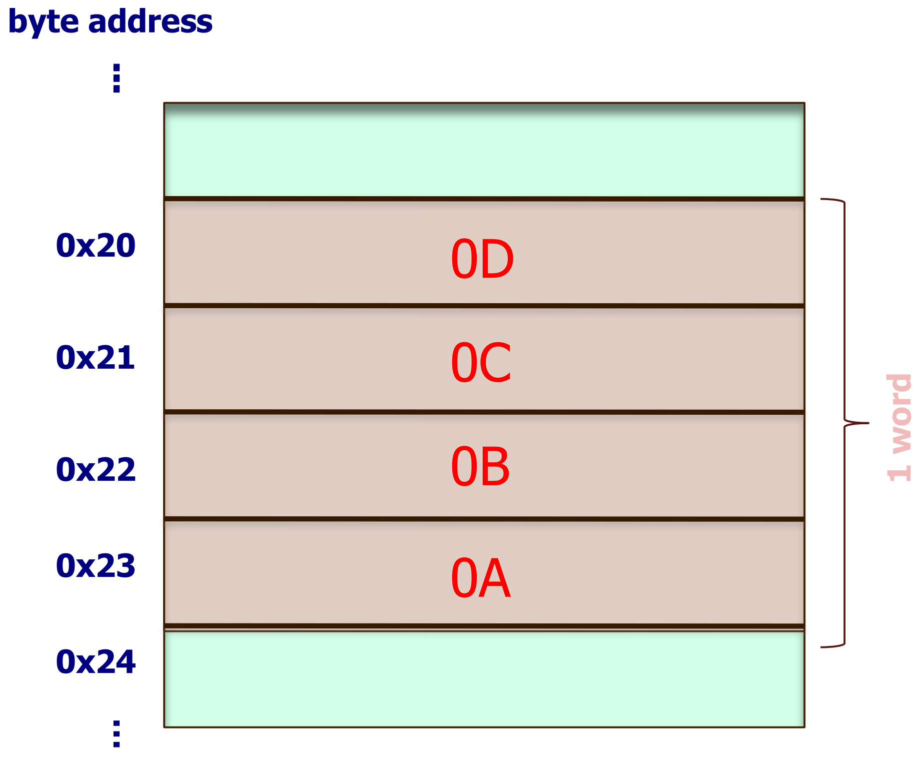


- In which order is the word stored in Memory?

Endianness: Little or Big?

- Big-Endian
  end of the word matches big addresses
- Little-Endian
  end of the word matches little addresses





Endianness (byte order): Little or Big?

- Some file formats and their endianess
  - **Adobe Photoshop** -- Big Endian
  - **BMP (Windows and OS/2 Bitmaps)** -- Little Endian
  - **GIF** -- Little Endian
  - **IMG (GEM Raster)** -- Big Endian
  - **JPEG** -- Big Endian
  - **PostScript** -- Not Applicable (text!)
  - **Microsoft RTF (Rich Text Format)** -- Little Endian
  - **TIFF** -- Both, Endian identifier encoded into file

Dealing with Constant

- To do the expression register1 = register2 +/- constant:

```
addi $t0, $s1, 8      # $t0 = $s1 + 8
addi $t0, $t0, -1      # $t0 = $t0 – 1 (no subi)
addi $t0, $t0, -1     # $t0 = $t0 - 1
```

- addi means add immediate (constant)
  - Constant part is always the last operand of this instruction

- Various ways to initialize a register with zero (or some constants):

```
i)   sub  $t0, $t0, $t0      # $t0 = 0 for sure
ii)  add  $t0, $zero, $zero  # same effect
iii) addi $t0, $zero, 5      # $t0 = 0 + 5 = 5
```

Memory vs. Register vs. Constant

- Major difference is the instruction's execution time
  - ① **Memory** is outside the processor; far from the processing unit
    - Memory operand takes a long time to load/store
  - ② **Register** is inside the processor; close to the processing unit
    - Register operand takes a short time to get to the value
  - ③ **Constant** already encoded in the instruction
    - Constant operand value is immediately available

A program is a mixture of these three types of operations

If you are to optimize the program running time, what should you do?

Focus on the Common Case

Design Principle #3

Make the common case fast

- Constant operands occur frequently!
- By including constants inside arithmetic instructions,
  - They are much faster than if constants were loaded from memory

Logical Operations – and, or, nor

- `and`, `or`, `nor`: bit-by-bit operation

bit 1	bit 2	and	or	nor
0	0	0	0	1
0	1	0	1	0
1	0	0	1	0
1	1	1	1	0

- **Example**
  - given register $t1 and $t2 , $\quad$ \$t1 = 0011 1100 0000 0000$_2$ \
    \$t2 = 0000 1101 0000 0000$_2$
  - `and   $t0, $t1, $t2` $\quad$ \$t0 = 0000 1100 0000 0000$_2$
  - `or    $t0, $t1, $t2` $\quad$ \$t0 = 0011 1101 0000 0000$_2$
  - `nor   $t0, $t1, $zero` $\quad$ \$t0 = 1100 0011 1111 1111$_2$

- `andi`: `and` with an immediate operand
- `ori`: `or` with an immediate operand

Logical Operations – shift

shift

- Move all the bits in a word to the left or right
- Filling the emptied bits with 0s
- Example

  `0000 0000 0000 0000 0000 0000 0000 1001` = 9_{10}

  shift left (<<) by 4

  `0000 0000 0000 0000 0000 0000 1001 0000` = 144_{10}

  shifting left by k bits gives the same result as multiplying by 2^k

MIPS shift instructions:

`sll` ('shift left logical'), `srl` ('shift right logical')

- Example

```
sll $t2, $s0, 4      # reg $s0 << 4 bits
```

What Have We Learned So Far about MIPS ISA?

Three types of instructions: arithmetic, logical, data transfer

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1, $s2, $s3	$s1 = $s2 + $s3	3 operands
	subtract	sub $s1, $s2, $s3	$s1 = $s2 - $s3	3 operands
	add immediate	addi $s1, $s2, 100	$s1 = $s2 + 100	2 operands, 1 constant
Logical	and	and $s1, $s2, $s3	$s1 = $s2 & $s3	3 operands, bit-by-bit and
	or	or $s1, $s2, $s3	$s1 = $s2 | $s3	3 operands, bit-by-bit or
	nor	nor $s1, $s2, $s3	$s1 = ~($s2 | $s3)	3 operands, bit-by-bit or
	and immediate	andi $s1, $s2, 100	$s1 = $s2 & 100	2 operands, 1 constant, bit-by-bit
	or immediate	ori $s1, $s2, 100	$s1 = $s2 | 100	2 operands, 1 constant, bit-by-bit
	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
Data transfer	load word	lw $s1, 100($s2)	$s1 = memory[$2+100]	Word from mem to reg
	store word	sw $s1, 100($s2)	Memory[$2+100] = $1	Word from reg to mem

Writing a Program in MIPS

- How can I declare an array / a variable in a MIPS program?
- How can I obtain the starting address of an array?
- How does the program run?

Sample Program

```
#############
# - We need to declare "variables" & "Arrays" used in the program in a
#   data segment.
# - The compiler recognize .data as the beginning of data segment
.data
h: .word 1 2 3 4  # h is an array of size 4, each element is a word (32 bit)
s: .word 5

# The 3 lines below let the system know the program begins here
.text
.globl __start
__start:

# Write your program code here
la $s0, h             # Obtain starting address of array h,  s0 = x (a constant)
lw $s1,8($s0)         # $s1 = content in memory address x + 8 = 3 = h[2]

la $s2, s
lw $s3, -12($s2)  # $s2 = content of address of s -12 = ?
sub $s3, $s3, $s1 # Q1: Guess what is the value of $s3 ?
sw $s3, 0($s0)    # Q2: How are the values of array h changed ?
```

How Does It Work?

- When the program is about to run, the data (variables, arrays) declared will be fed into memory consecutively

```
h: .word 1 2 3 4 # h is an array of size 4
s: .word 5
```

	Address	Value	Array element
h →	X -th byte	1	h[0]
	X+4 -th byte	2	h[1]
	X+8 -th byte	3	h[2]
	X+12 -th byte	4	h[3]
s →	X+16 -th byte	5	

- h & s are called “labels”, they can be viewed as the bookmarks of the program
- When la $s0, h is executed, the address (in byte) referenced by h will be assigned to register $s0
- e.g. if X = 10000, then $s0 = 10000. This means the values of the array h store between the 10000th and 10015th byte of the memory

2. Implement MIPS Instructions

Representing Instructions in the Computer

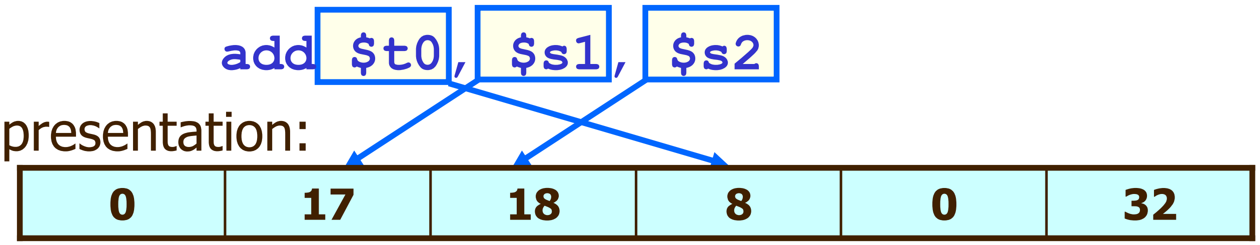
How does the computer "see" the instructions?

- Machine language or machine code; represented as binary numbers
  - Numeric data are kept in computer as a series of **high** & **low** electronic signals – base 2 or binary numbers
  - A binary digit, or bit, is the basic unit of digital computing

What is the format of the machine code?

- All MIPS instructions are 32-bit long, broken up into a number of fields

- Example: `add $t0, $s1, $s2`
  - Decimal representation:



0	**17**	**18**	**8**	**0**	**32**

  - Binary representation:

000000	**10001**	**10010**	**01000**	**00000**	**100000**
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Instruction Formats

- All MIPS instructions have fixed length, but different instructions may have different formats
- Three types of instruction formats in MIPS
  - R-type or R-format for register
  - I-type or I-format for immediate
  - J-type or J-format for jump
- Each format is assigned a distinct set of values for the 1st field
  - Hardware can interpret the instruction just by examining this field
  - This field is so-called opcode
- Using multiple formats complicates hardware design, but complexity can be reduced by keeping the formats similar (will see in next slides)

R-type Instruction Format

R-type or R-format

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- **Instruction fields**: (6 fields)
  - **op**: basic operation of instruction, traditionally called **opcode**
  - **rs**: first register source operand
  - **rt**: second register source operand
  - **rd**: register destination operand, which gets result of operation
  - **shamt**: shift amount (number of positions to shift)
  - **funct**: function code selecting the specific variant of the opcode

I-type Instruction Format

I-type or **I-format**

op	rs	rt	const or address
6 bits	5 bits	5 bits	16 bits

- **Instruction fields**:
  - **op**: as before
  - **rs**: base register
  - **rt**: register source operand (for `sw`)
    or destination operand (for `lw`)
  - **address**: 16-bit address offset from the starting address
    - needed for data transfer instructions
    - 5-bit field in the R-format is too small for specifying the offset for reasonably sized arrays.

Fixed vs. Variable Instruction Length

Design Principle #4

Good design demands good compromises

- Ways to encode instructions:
  - **Variable** length *or* **fixed** length

- How to choose?
  - Use **variable length** to optimize code size (i.e. to save storage)
  - Use **fixed length** to optimize performance and reduce complexity

- Compromise MIPS chose is to keep all instructions the same length
  - Why?
  - Hardware to fetch & decode an instruction is simpler and faster

MIPS Instruction Encoding

Instruction	Type	op	rs	rt	rd	shamt	funct	const/address
add	R	0	reg	reg	reg	0	32_{10}	-
sub	R	0	reg	reg	reg	0	34_{10}	-
and	R	0	reg	reg	reg	0	36_{10}	-
or	R	0	reg	reg	reg	0	37_{10}	-
sll	R	0	0	reg	reg	constant	0	-
srl	R	0	0	reg	reg	constant	2_{10}	-
addi	I	8_{10}	reg	reg	-	-		constant
lw	I	35_{10}	reg	reg	-	-		address
sw	I	43_{10}	reg	reg	-	-		address

- **reg**: a register number between 0 and 31
- **const/address**: a constant or a 16-bit address (offset)

- e.g. both `add` and `sub` have the same value in **op** field but different values (32 for `add`; 34 for `sub`) in the **funct** field.

Naming the Registers in MIPS

- Not by alphabet, but by number!

Symbolic 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

Notes:

- register 1, called $at, is reserved for the assembler
- register 26-27, called $k0-$k1, are reserved for the operating system

Example - Problem

- **Description**:
  - Suppose `$t1` stores the base address of array `A` and `$s2` is associated with `h`, the following C assignment statement

    ```
    A[300] = h + A[300];
    ```

    is compiled into

    ```
    lw  $t0, 1200($t1)   # $t0 gets A[300]
    add $t0, $s2, $t0    # $t0 gets h + A[300]
    sw  $t0, 1200($t1)   # A[300] gets h + A[300]
    ```

- **Problem to solve**:
  - What is the MIPS machine code for these three instructions?

Example - Answer

```
lw  $t0, 1200($t1)
add $t0, $s2, $t0
sw  $t0, 1200($t1)
```

- **Decimal representation**:

op	rs	rt	rd	address /shamt	funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

- **Binary representation**:

100011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000 0100 1011 0000		

Other Important MIPS Instructions

Unsigned arithmetic

- `addu $d, $s, $t`
- `subu $d, $s, $t`

Load/store a byte

- `lb $t, offset($s)`
- `sb $t, offset($s)`

Logical operation

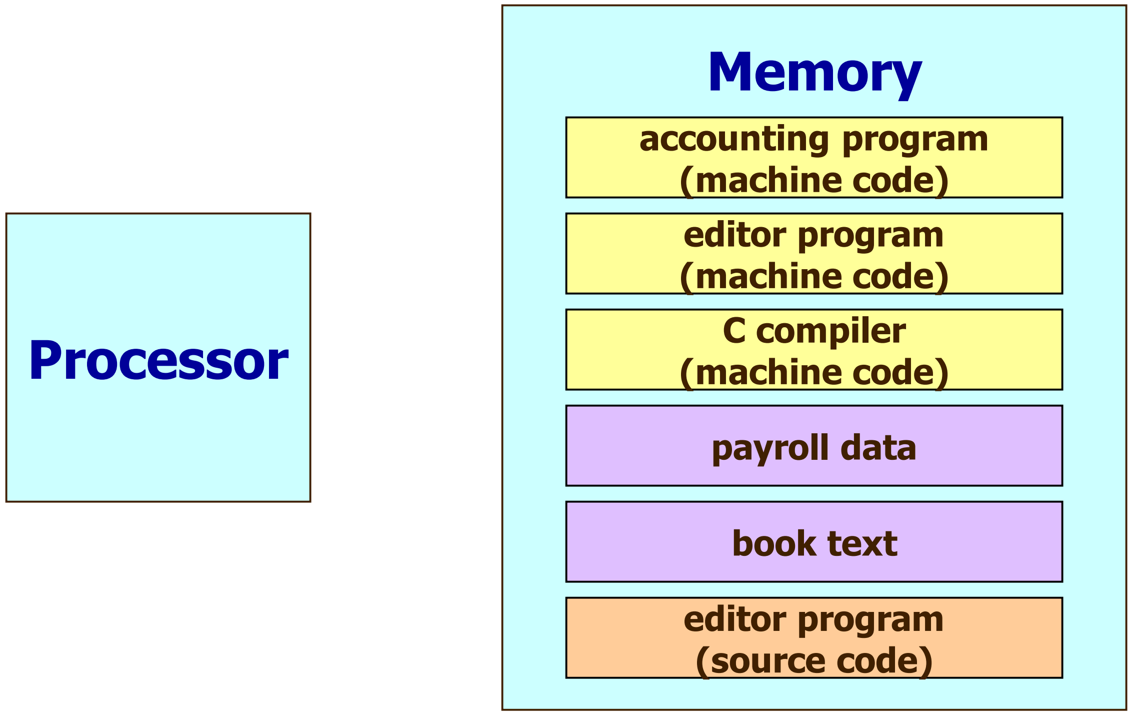
```
xor $d, $s, $t          # $d = $s ^ $t
xori $t, $s, immd       # $t = $s ^ imm
```

Shift left/right logical variable

```
sllv $d, $t, $s         # $d = $t << $s
srlv $d, $t, $s         # $d = $t >> $s
```

The Stored-Program Concept

- Today's computers are built on two key principles
  - **Instructions** are represented as **numbers**.
  - **Programs** can be stored in **memory** to be read or written, just like **numeric data**.


Processor
Memory
accounting program (machine code)
editor program (machine code)
C compiler (machine code)
payroll data
book text
editor program (source code)