Computer Architectures

Machine language(1)

Zhu Wen-Jun

Institute of Computer

Levels of Representation/Interpretation

High Level Language Program (e.g., C)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

Compiler

Assembly Language Program (e.g., MIPS)

```
lw    $t0, 0($2)
lw    $t1, 4($2)
sw    $t1, 0($2)
sw    $t0, 4($2)
```

Anything can be represented as a *number*, i.e., data or instructions

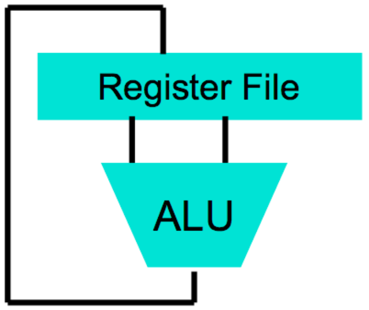
Assembler

Machine Language Program (MIPS)

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

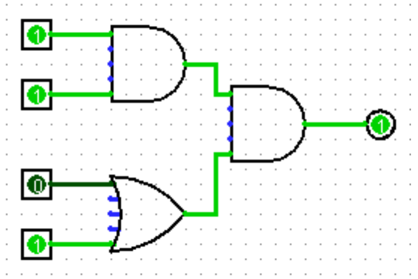
Machine Interpretation

Hardware Architecture Description (e.g., block diagrams)



Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)



Outline

- Machine Language
- Register
- Instruction and Immediate
- Data Transfer Instruction
- Judge Instruction

Outline

- Machine Language
- Register
- Instruction and Immediate
- Data Transfer Instruction
- Judge Instruction

Machine Language (1/2)

- "Word" a computer understands: *instruction*
- Vocabulary of all "words" a computer understands: *instruction set architecture (ISA)*

- Why might you want the same ISA? Why might you want different ISAs?
  - e.g. iPhone and iPad use the same
  - e.g. iPhone and Macbook use different

Machine Language (2/2)

- Single ISA
  - Leverage common compilers, operating systems, etc.
  - BUT fairly easy to retarget these for different ISAs (e.g. Linux, gcc)
- Multiple ISAs
  - Specialized instructions for specialized applications
  - Different tradeoffs in resources used (e.g. functionality, memory demands, complexity, power consumption, etc.)
  - Competition and innovation is good, especially in emerging environments (e.g. mobile devices)

Why Study Assembly?

- Understand computers at a deeper level
  - Learn to write more compact and efficient code
  - Can sometimes hand optimize better than a compiler
- More sensible for minimalistic applications
  - e.g. distributed sensing and systems
  - Eliminating OS, compilers, etc. reduce size and power consumption
  - Embedded computers outnumber PCs!

Reduced Instruction Set Computing

- The early trend was to add more and more instructions to do elaborate operations – this became known as *Complex Instruction Set Computing* (CISC)
- Opposite philosophy later began to dominate: *Reduced Instruction Set Computing* (RISC)
  - Simpler (and smaller) instruction set makes it easier to build fast hardware
  - Let software do the complicated operations by composing simpler ones

RISC Design Principles

- Basic RISC principle: “A simpler CPU (the hardware that interprets machine language) is a faster CPU” (CPU → *Core*)
- Focus of the RISC design is reduction of the number and complexity of instructions in the ISA
- A number of the more common strategies include:
  - Fixed instruction length, generally a single word;
    Simplifies process of fetching instructions from memory
  - Simplified addressing modes;
    Simplifies process of fetching operands from memory
  - Fewer and simpler instructions in the instruction set;
    Simplifies process of executing instructions
  - Only load and store instructions access memory;
    E.g., no add memory to register, add memory to memory, etc.
  - *Let the compiler do it.* Use a good compiler to break complex high-level language statements into a number of simple assembly language statements

Mainstream ISAs

- Intel 80x86
  - Used in Macbooks and PCs
  - Found in Core i3, Core i5, Core i7, etc.
- Advanced RISC Machine (ARM)
  - Smart phone-like devices: iPhone, iPad, iPod, etc.
  - The most popular RISC (20x more common than 80x86)
- IBM/Motorola/Apple PowerPC
  - Airborne equipment: Flight Control, Radar, etc.
  - Engine controller
  - Embedded devcies: Network switch/router, etc

Instruction Set in CS 61C

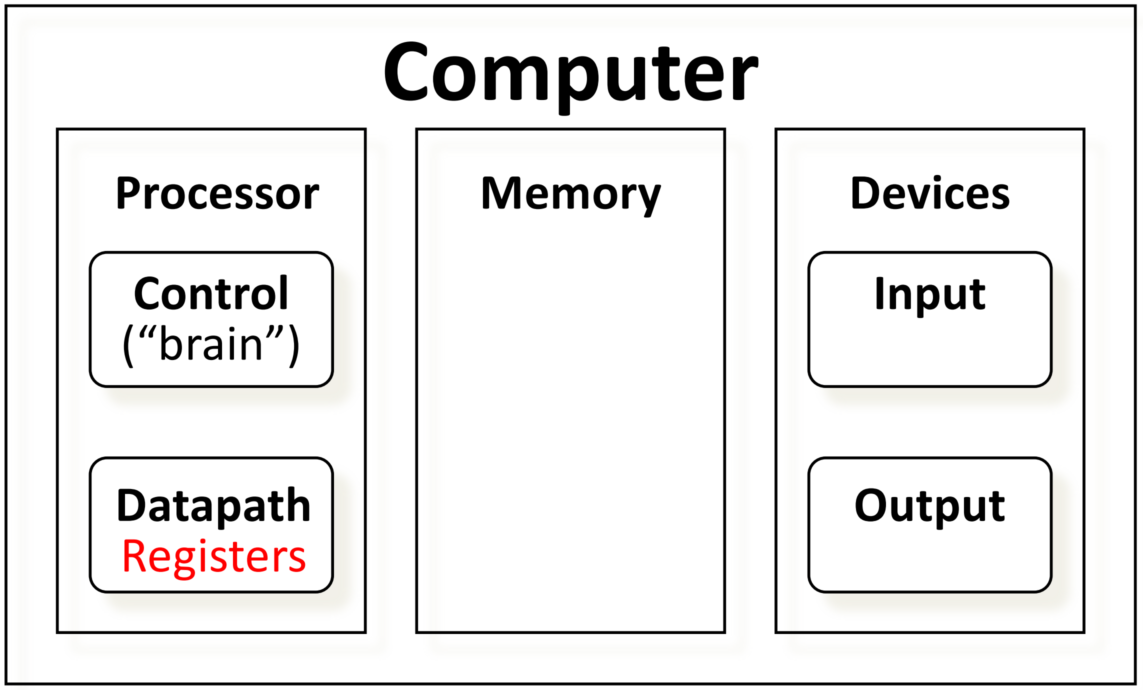
- MIPS
  - Invented by John Hennessy @ Stanford
    - *(Why not Berkeley/Sun RISC invented by Dave Patterson? Ask him!)*
  - MIPS is a real world ISA
    - Standard instruction set for networking equipment
- Elegant example of *Reduced Instruction Set Computer* (RISC) instruction set

Outline

- Machine Language
- Register
- Instruction and Immediate
- Data Transfer Instruction
- Judge Instruction

Five Components of a Computer

- We begin our study of how a computer works!
  - Control
  - Datapath
  - Memory
  - Input
  - Output



- Registers are part of the Datapath

Computer Hardware Operands

- In high-level languages, number of variables limited only by available memory
- ISAs have a fixed, small number of operands called registers
  - Special locations built directly into hardware
  - **Benefit:** Registers are EXTREMELY FAST (faster than 1 billionth of a second)
  - **Drawback:** Operations can only be performed on these predetermined number of registers

MIPS Registers

- MIPS has 32 registers
  - Each register is 32 bits wide and hold a word
- Tradeoff between speed and availability
  - Smaller number means faster hardware but insufficient to hold data for typical C programs
- Registers have no type (C concept); the operation being performed determines how register contents are treated

MIPS Registers

- Register denoted by ‘`$`’ can be referenced by number (`$0-$31`) or name:
  - Registers that hold programmer variables:

    $s0-$s7 $\longleftrightarrow$ $16-$23
  - Registers that hold temporary variables:

    $t0-$t7 $\longleftrightarrow$ $8-$15

    $t8-$t9 $\longleftrightarrow$ $24-$25
  - You’ll learn about the other 14 registers later
- In general, using register names makes code more readable

Outline

- Machine Language
- Register
- Instruction and Immediate
- Data Transfer Instruction
- Judge Instruction

MIPS Instructions

- Instruction Syntax is rigid:

```
op dst, src1, src2
```

  - 1 operator, 3 operands
    - `op` = operation name ("operator")
    - `dst` = register getting result ("destination")
    - `src1` = first register for operation ("source 1")
    - `src2` = second register for operation ("source 2")
- Keep hardware simple via regularity

MIPS Instructions

- One operation per instruction, at most one instruction per line
- Assembly instructions are related to C operations (`=`, `+`, `-`, `*`, `/`, `&`, `|`, etc.)
  - Must be, since C code decomposes into assembly!
- A single line of C may break up into several lines of MIPS

MIPS Instructions Example

- Your very first instructions!
  (assume here that the variables `a`, `b`, and `c` are assigned to registers `$s1`, `$s2`, and `$s3`, respectively)
- Integer Addition (`add`)
  - C: `a = b + c`
  - MIPS: `add  $s1, $s2, $s3`
- Integer Subtraction (`sub`)
  - C: `a = b - c`
  - MIPS: `sub  $s1, $s2, $s3`

MIPS Instructions Example

- Suppose `a`→`$s0`, `b`→`$s1`, `c`→`$s2`, `d`→`$s3`, and `e`→`$s4`. Convert the following C statement to MIPS:

```
a = (b + c) – (d + e);
```

```
add $t1, $s3, $s4
add $t2, $s1, $s2
sub $s0, $t2, $t1
```

Ordering of instructions matters (must follow order of operations)

Utilize temporary registers

Comments in MIPS

- Comments in MIPS follow hash mark (#) until the end of line
  - Improves readability and helps you keep track of variables/registers!

```
add $t1, $s3, $s4 # $t1=d+e
add $t2, $s1, $s2 # $t2=b+c
sub $s0, $t2, $t1 # a=(b+c)-(d+e)
```

The Zero Register

- Zero appears so often in code and is so useful that it has its own register!
- Register zero (`$0` or `$zero`) always has the value 0 and cannot be changed!
  - i.e. any instruction with `$0` as `dst` has no effect
- Example Uses:
  - `add  $s3,  $0,  $0  # c=0`
  - `add  $s1, $s2,  $0  # a=b`

Immediates

- Numerical constants are called immediates
- Separate instruction syntax for immediates:

```
    opi dst, src, imm
```

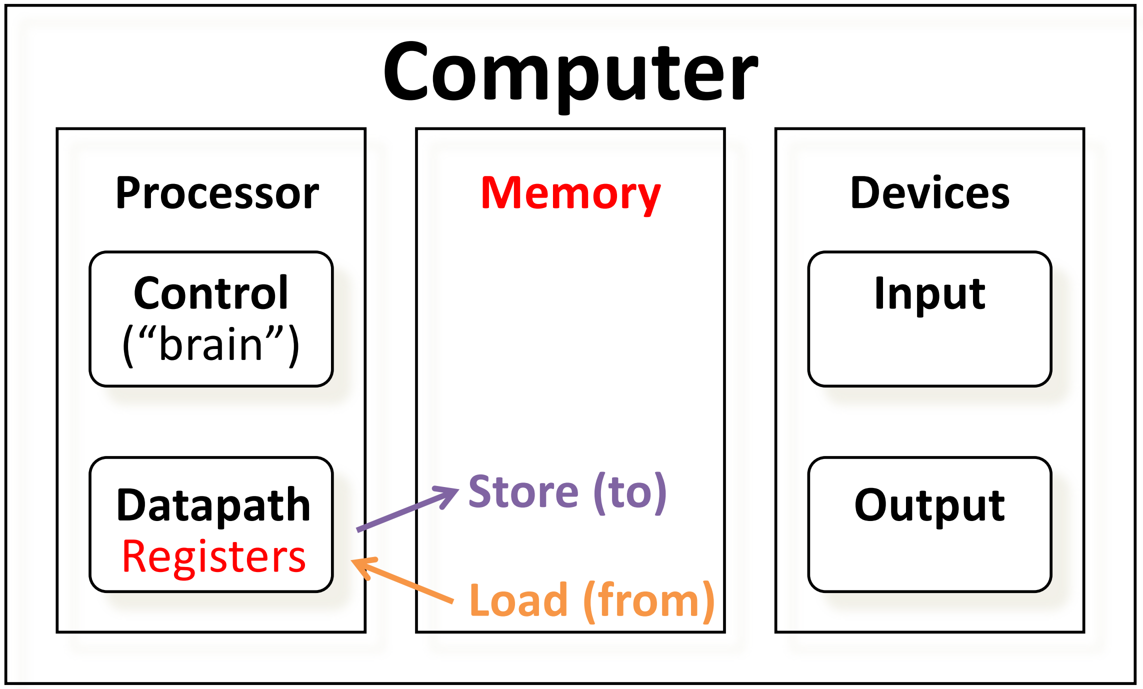
  - Operation names end with ‘`i`’, replace 2nd source register with an immediate (check Green Sheet for un/signed)
- Example Uses:
  - `addi $s1, $s2, 5   # a=b+5`
  - `addi $s3, $s3, 1   # c++`
- Why no `subi` instruction?

Outline

- Machine Language
- Register
- Instruction and Immediate
- Data Transfer Instruction
- Judge Instruction

Five Components of a Computer

- Data Transfer instructions are between registers (Datapath) and Memory
  - Allow us to fetch and store operands in memory



Data Transfer

- C variables map onto registers;
  What about large data structures like arrays?
  - Don't forget *memory*, our one-dimensional array indexed by addresses starting at 0
- MIPS instructions only operate on registers!
- Specialized data transfer instructions move data between registers and memory
  - Store: register TO memory
  - Load: register FROM memory

Data Transfer

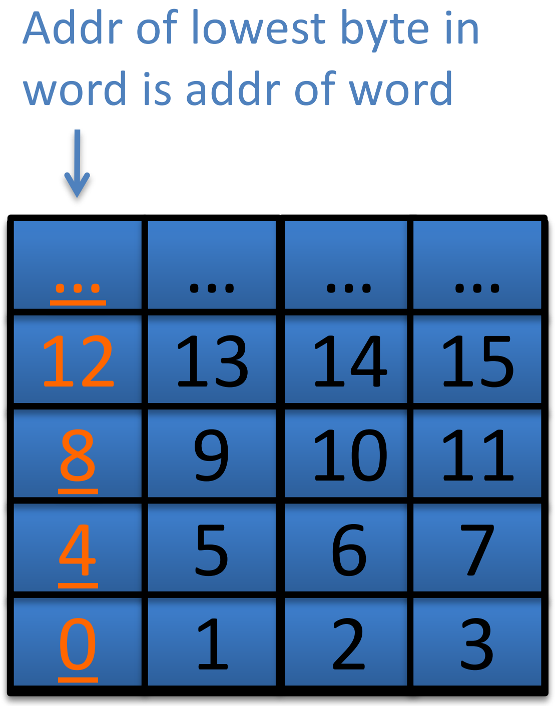
- Instruction syntax for data transfer:

```
op reg, off(bAddr)
```

  - `op` = operation name ("operator")
  - `reg` = register for operation source or destination
  - `bAddr` = register with pointer to memory ("base address")
  - `off` = address offset (immediate) in bytes ("offset")
- Accesses memory at address `bAddr+off`
- **Reminder:** A register holds a word of raw data (no type) – make sure to use a register (and offset) that point to a valid memory address

Memory is Byte-Addressed

- What was the smallest data type we saw in C?
  - A char, which was a *byte* (8 bits)
  - Everything in multiples of 8 bits (e.g. 1 word = 4 bytes)
- Memory addresses are indexed by *bytes*, not words
- Word addresses are 4 bytes apart
  - Word addr is same as left-most byte
  - Offsets are multiples of 4 to be "word-aligned"
- Pointer arithmetic not done for you in assembly
  - Must take data size into account yourself



Data Transfer Instructions

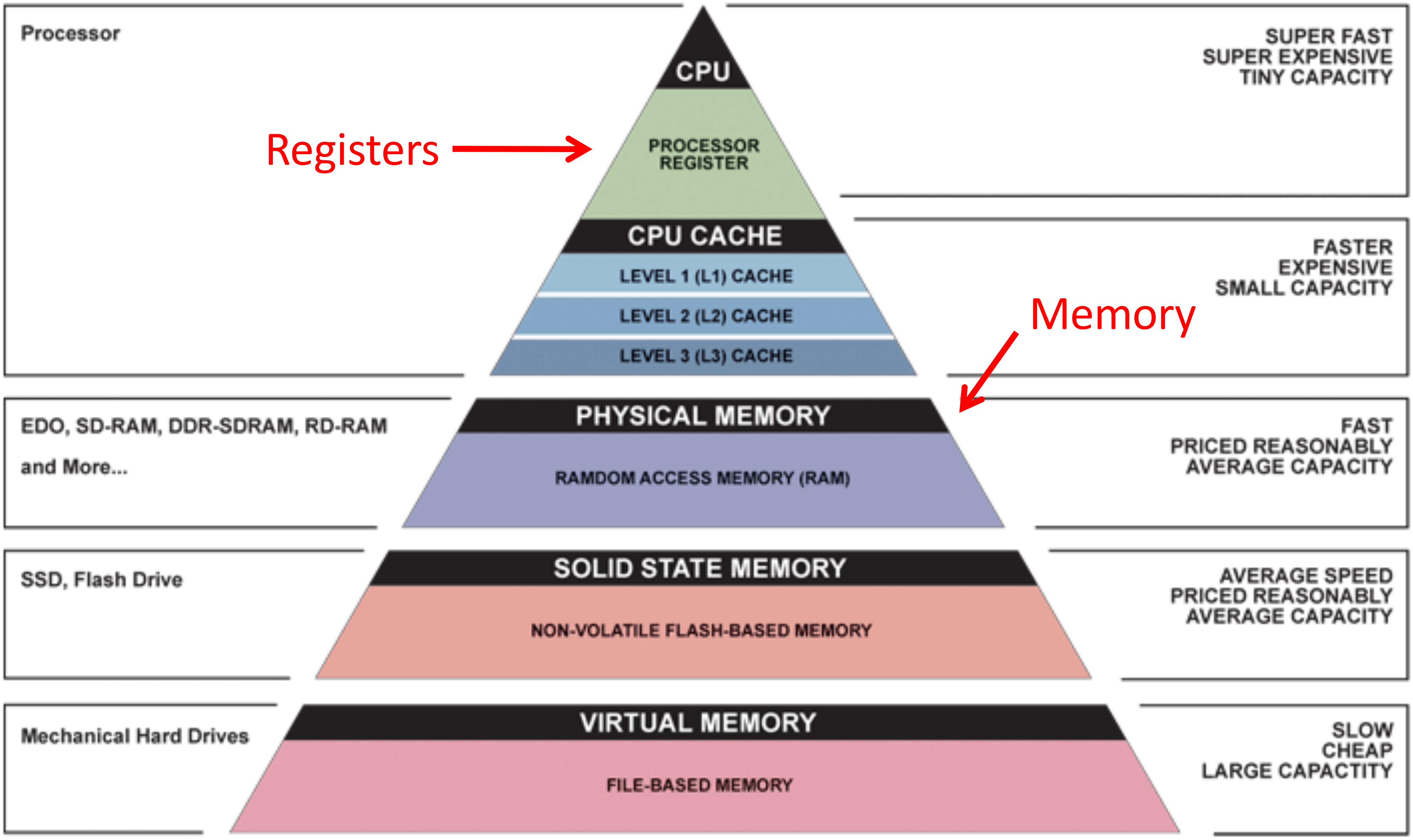
- Load Word (`lw`)
  - Takes data at address `bAddr+off` FROM memory and places it into `reg`
- Store Word (`sw`)
  - Takes data in `reg` and stores it TO memory at address `bAddr+off`
- Example Usage:

```
# addr of int A[] -> $s3, a -> $s0
lw  $t0,12($s3) # $t0=A[3]
add $t0,$s0,$t0 # $t0=A[3]+a
sw  $t0,40($s3) # A[10]=A[3]+a
```

Registers vs. Memory

- What if more variables than registers?
  - Keep most frequently used in registers and move the rest to memory (called *spilling* to memory)
- Why not all variables in memory?
  - Smaller is faster: registers 100-500 times faster
  - Registers more versatile
    - In 1 arithmetic instruction: read 2 operands, perform 1 operation, and 1 write
    - In 1 data transfer instruction: 1 read/write, no operation

Great Idea #3: Principle of Locality/ Memory Hierarchy

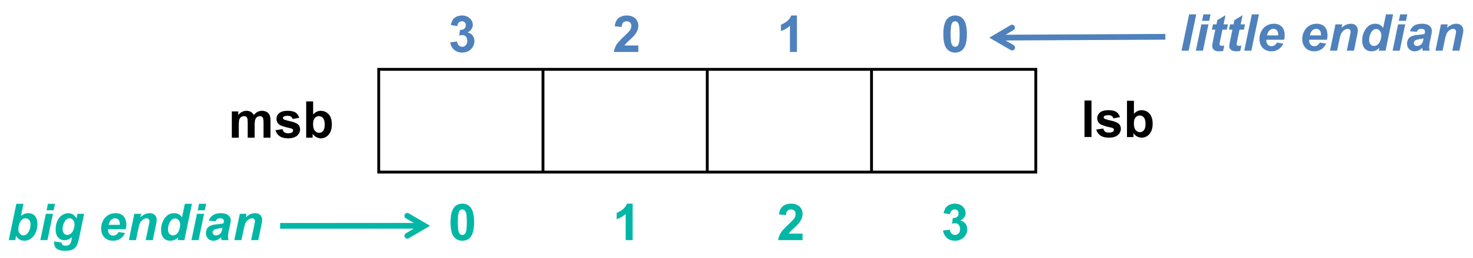


Question: Which of the following is TRUE?

- ☐ `add $t0,$t1,4($t2)` is valid MIPS
- ☐ Can byte address 8GB with a MIPS word
- ☐ `off` must be a multiple of 4 for `lw $t0,off($s0)` to be valid
- ☐ If MIPS halved the number of registers available, it would be twice as fast

Endianness

- Big Endian: Most-significant byte at least address of word
  - word address = address of most significant byte
- Little Endian: Least-significant byte at least address of word
  - word address = address of least significant byte



- MIPS is bi-endian (can go either way)
  - Using MARS simulator in lab, which is little endian
- Why is this confusing?
  - Data stored in reverse order than you write it out!
  - Data `0x01020304` stored as `04 03 02 01` in memory

Increasing address

Outline

- Machine Language
- Register
- Instruction and Immediate
- Data Transfer Instruction
- Judge Instruction

Computer Decision Making

- In C, we had *control flow*
  - Outcomes of comparative/logical statements determined which blocks of code to execute
- In MIPS, we can't define blocks of code; all we have are labels
  - Defined by text followed by a colon (e.g. `main:`) and refers to the instruction that follows
  - Generate flow control by jumping labels
  - C has these too, but they are considered bad style

Decision Making Instructions

- Branch If Equal (`beq`)
  - `beq reg1,reg2,label`
  - If value in `reg1` = value in `reg2`, go to `label`
- Branch If Not Equal (`bne`)
  - `bne reg1,reg2,label`
  - If value in `reg1` ≠ value in `reg2`, go to `label`
- Jump (`j`)
  - `j label`
  - Unconditional jump to `label`

Breaking Down the If Else

C Code:

```
if(i==j) {
  a = b  /* then */
} else {
  a = -b /* else */
}
```

In English:

- If TRUE, execute the <u>THEN</u> block
- If FALSE, execute the <u>ELSE</u> block

MIPS (beq):

```
# i→$s0, j→$s1
# a→$s2, b→$s3

beq $s0,$s1,???
???     ⟵ This label unnecessary
sub $s2, $0, $s3
j   end
then:
add $s2, $s3, $0
end:
```

Breaking Down the If Else

C Code:

```
if(i==j) {
  a = b  /* then */
} else {
  a = -b /* else */
}
```

In English:

- If TRUE, execute the THEN block
- If FALSE, execute the ELSE block

MIPS (bne):

```
# i→$s0, j→$s1
# a→$s2, b→$s3

bne $s0,$s1,???
???
add $s2, $s3, $0
j   end
else:
sub $s2, $0, $s3
end:
```

Loops in MIPS

- There are three types of loops in C:
  - `while`, `do…while`, and `for`
  - Each can be rewritten as either of the other two, so the same concepts of decision-making apply
- You will examine how to write these in MIPS in discussion
- **Key Concept:** Though there are multiple ways to write a loop in MIPS, the key to decision-making is the conditional branch

Summary

- Computers understand the *instructions* of their *ISA*
- RISC Design Principles
  - Smaller is faster, keep it simple
- MIPS Registers: `$s0-$s7`, `$t0-$t9`, `$0`
- MIPS Instructions
  - Arithmetic: `add`, `sub`, `addi`
  - Data Transfer: `lw`, `sw`, `lb`, `sb`, `lbu`
  - Branching: `beq`, `bne`, `j`
- Memory is byte-addressed

Homework

- 《Computer Organization and Design》
- WORD：2.1、2.4