MIPS Instructuction Set Architecture 1

Kuliah #4-6, 2019-2020/1

CSCM601252 – **Introduction to Computer Organization**

Instructor: Erdefi Rakun, Adhi Yuniarto

Fasilkom UI

Outline

- Instruction set architecture
  - RISC
  - CISC
- Simple MIPS Instructions
  - Addition
  - Subtraction
  - Constant/Immediate Operands
  - Register Zero
  - Logical Operations
  - MIPS Opcodes
  - Memory Access Instructions: Load/Store
  - Arrays
  - Large Constants
  - Branch Instructions
  - Inequalities
  - Array and Loop

Note: These slides are taken from Aaron Tan's slide

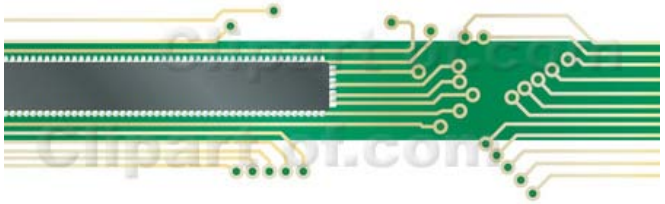
Instruction Set Architecture (1/5)

- Instruction Set Architecture (ISA): an abstraction on the interface between the hardware and the low-level software.

Software
(to be translated to the instruction set)

Instruction set architecture

Hardware
(implementing the instruction set)



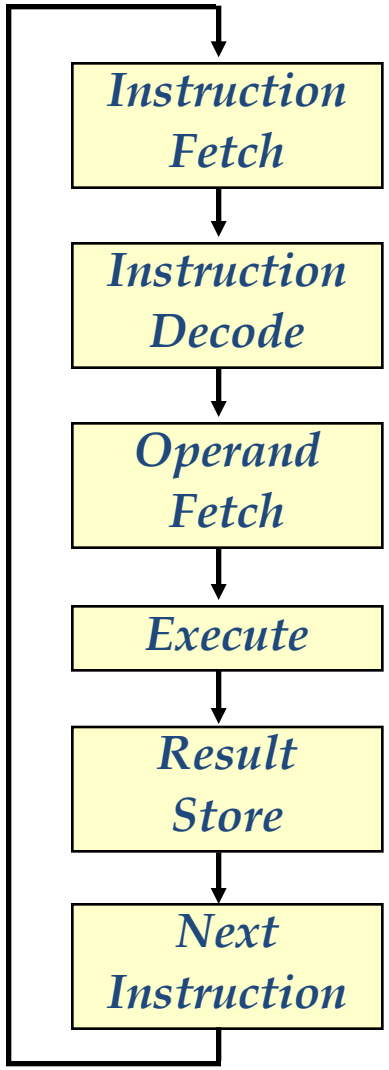
Instruction Set Architecture (2/5)

- The ISA includes everything programmers need to know to make the machine code work correctly.
- It allows computer designers to talk about functions independently from the hardware that performs them.
- This abstraction allows many implementations of varying cost and performance to run identical software.
  - Example: Intel x86/IA-32 ISA has been implemented by a range of processors starting from 80386 [1985] to Pentium 4 [2005]
  - Other companies such as AMD and Transmeta have implemented IA-32 ISA as well
  - A program compiled for IA-32 ISA can execute on any of these implementations

Instruction Set Architecture (3/5)

- ISA is determined by
  - Organization of programmable storage.
  - Data types and data structures: encoding and representations.
  - Instruction formats.
  - Instruction (or operation code, opcode) set.
  - Modes of addressing and accessing data items and instructions.
  - Exceptional conditions.

Instruction Set Architecture (4/5)



- Instruction format or encoding
  - how is it decoded?
- Location of operands and result
  - where other than memory?
  - how many explicit operands?
  - how are memory operands located?
  - which can or cannot be in memory?
- Data type and size
- Operations
  - what are supported?
- Successor instruction
  - jumps, conditions, branches
- Fetch-decode-execute is always done for any instruction!

Instruction Set Architecture (5/5)

- Instruction set: language of the machine.
- More primitive than high-level languages (eg: no sophisticated control flow).
- More restrictive instructions.
- We will study MIPS instruction set, used by NEC, Nintendo, Silicon Graphics, Sony.

Assembly Language

- Machine code
  - Instructions are represented in binary
  - 1000110010100000 is an instruction that tells one computer to add two numbers
  - Hard and tedious for programmer
- Assembly language
  - Symbolic version of machine code
  - Human readable
  - **add A, B** is equivalent to 1000110010100000
  - Assembler translates from assembly language to machine code
  - Assembly can provide 'pseudo-instructions'. Example: In MIPS, "move $t0, $t1" is a pseudo-instructions; the actual instruction is "add $t0, $t1, $zero".
  - When considering performance, only real instructions are counted.

RISC vs CISC

- Complex Instruction Set Computer (CISC) e.g. x86
  - Single instruction performs complex operation
    - VAX architecture had an instruction to multiply polynomials
  - Smaller program size as memory was premium
  - Complex implementation, no room for hardware optimization
- Reduced Instruction Set Computer (RISC) e.g. MIPS
  - Keep the instruction set small and simple, makes it easier to build/optimize hardware
  - Burden on software to combine simpler operations to implement high-level language statements
- Pentium 4 decomposes many x86 instructions into a series of internal *micro-operations* that are executed by the processor core
  - CISC ISA but internal RISC implementation

MIPS Assembly Language

- In MIPS assembly language, each instruction executes a simple command
- Each line of assembly code contains at most 1 instruction
- Instructions are related to operations (+, -, *, /, =) in C/Java
- # is used for comments
  - Anything from # mark to end of line is a comment and will be ignored

```
add $t0, $s1, $s2  # tmp = b + c
sub $s0, $t0, $s3  # a = tmp - d
```

Acknowledgement: The slides starting from here are taken from Dr Tulika's CS1104 materials.

Arithmetic: Addition

- Addition in assembly

  C: **a = b + c;**

  MIPS: **add $s0, $s1, $s2**

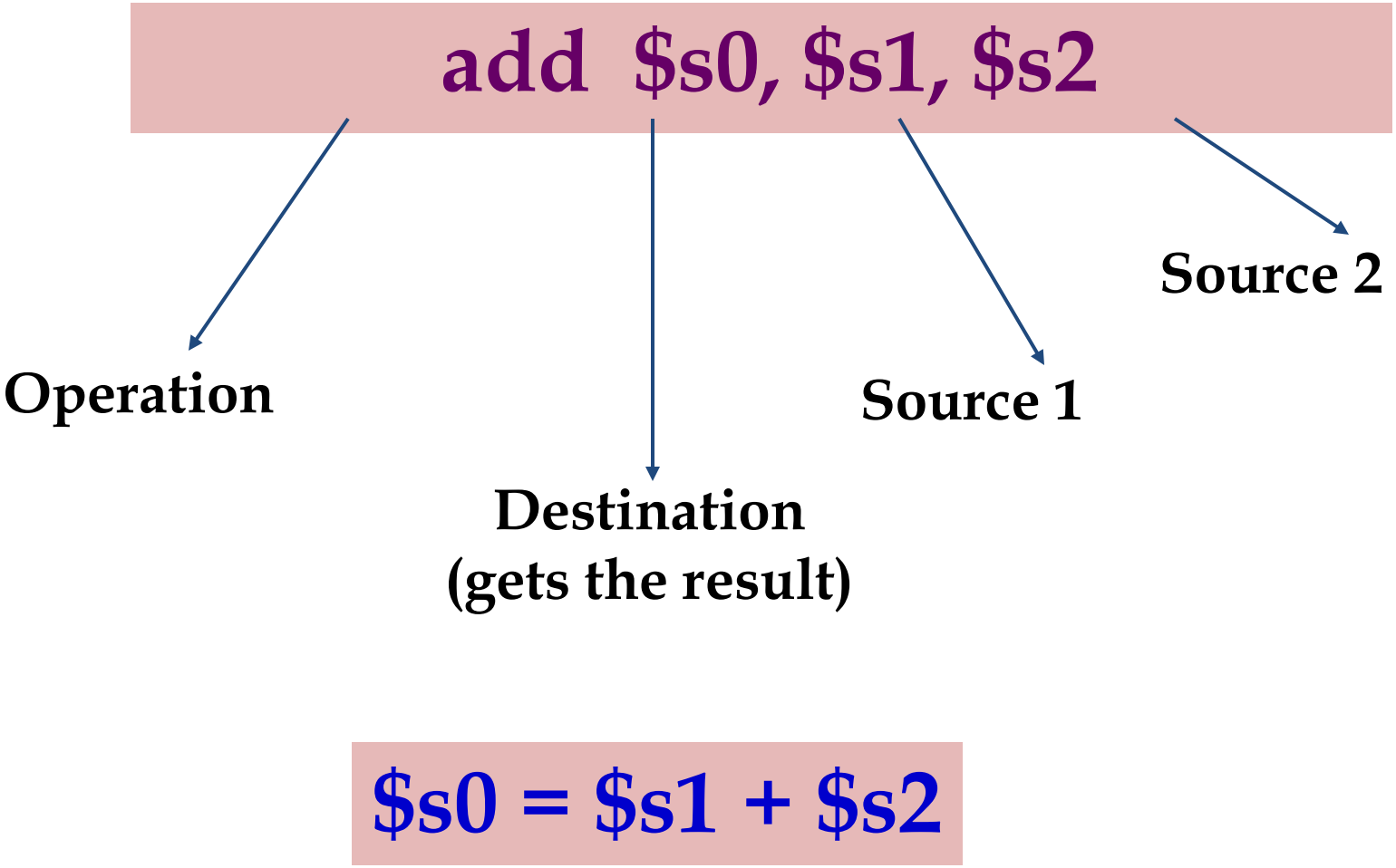
  $s0 → variable **a**

  $s1 → variable **b**

  $s2 → variable **c**

- Natural number of operands for an instruction is 3 (2 sources + 1 destination)
- Why?
  - Keep the hardware simple
  - Design principle: Simplicity favors regularity

Instruction Syntax



$s0 = $s1 + $s2

MIPS Registers

- MIPS assembly language: 32 registers, referred to by a number ($0, $1, …, $31) or a name (eg: $a0, $t1).

Name	Register number	Usage
$zero	0	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	Program variables

Name	Register number	Usage
$t8-$t9	24-25	More temporaries
$gp	28	Global pointer
$sp	29	Stack pointer
$fp	30	Frame pointer
$ra	31	Return address

$at (register 1) is reserved for the assembler.
$k0-$k1 (registers 26-27) are reserved for the operation system.

Arithmetic: Subtraction

- Subtraction in assembly

  C: **a = b – c;**

  MIPS: **sub $s0, $s1, $s2**

  $s0 → variable **a**

  $s1 → variable **b**

  $s2 → variable **c**

  $s0 = $s1 - $s2

- Positions of **$s1** and **$s2** (i.e., source1 and source2) are important for subtraction

Complex Statements (1/2)

- C statement

  a = b + c - d;

- A single instruction can handle at most two source operands.
- Break it up into multiple instructions and use temporary register **$t0 … $t7**

```
add $t0, $s1, $s2  # tmp = b + c
sub $s0, $t0, $s3  # a = tmp - d
```

- A single line of C statement may break up into several lines of MIPS assembly
- Who is doing this break up when you write high-level language code?

Complex Statements (2/2)

- C statement

  f = (g + h) – (i + j);

- Break it up into multiple instructions
- Replace variables by registers **$s0 … $s7**
- Use two temporary registers **$t0, $t1**

```
add $t0, $s1, $s2  # tmp0 = g + h
add $t1, $s3, $s4  # tmp1 = i + j
sub $s0, $t0, $t1  # a = tmp0 – tmp1
```

TRY IT YOURSELF

- **z = a + b + c + d;**
  - **a:$s0 b:$s1 c:$s2 d:$s3 z:$s4**

- **z = (a – b) + c;**
  - **a:$s0 b:$s1 c:$s2 z:$s4**

Constant/Immediate Operands

- Immediates are numerical constants
- They appear often in operations; so there is special instruction for them
- "Add immediate" (addi)

  C: **a = a + 4;**

  MIPS: **addi $s0, $s0, 4 # $s0 = $s0 + 4**

- Syntax is similar to **add** instruction; but source2 is a constant instead of register
- Design principle: Make common case fast
- No **subi** instruction in MIPS – why?

Register Zero

- One particular immediate, the number zero (0), appears very often in code
- Define register zero (**$0** or **$zero**) to always have the value 0

C: `f = g;`

MIPS: `add $s0, $s1, $zero`

where MIPS registers **$s0** and **$s1** are associated with variables **f** and **g**

- This instruction

`add $zero, $zero, $s0`

does not do anything!

Logical Operations

- Arithmetic instructions view content of a register as a single quantity (signed or unsigned integer)
- New perspective: View register as 32 raw bits rather than as a single 32-bit number
- Consequence: Useful to operate on individual bytes or bits within a word

Logical operation	C operator	Java operator	MIPS instruction
Shift Left	<<	<<	sll
Shift right	>>	>>, >>>	srl
Bitwise AND	&	&	and, andi
Bitwise OR	|	|	or, ori
Bitwise NOT	~	~	nor

Shift Instructions (1/2)

- Opcode: **sll** (shift left logical)
  - Move all the bits in a word to the left by a number of bits; fill the emptied bits with zeroes
- Example: register **$s0** contains

0000 1000 0000 0000 0000 0000 0000 1001

```
sll $t2, $s0, 4  # $t2 = $s0<<4
```

shifts left by 4 bits results in content of register **$t2**

1000 0000 0000 0000 0000 0000 1001 0000

- Q: Shifting left by *n* bits gives the same result as multiplying by what value? Answer:

Shift Instructions (2/2)

- Opcode: **srl** (shift right logical)
  - Shifts right and fills emptied bits with zeroes
- Q: Shifting right by *n* bits is the same as what mathematical operation? Answer:
- Shifting is faster than multiplication/division
- Good compiler translates such multiplication/division into shift instructions:

C: `a = a * 8;`
MIPS: `sll $s0, $s0, 3`

Bitwise AND Instruction (1/2)

- AND: bitwise operation that leaves a 1 only if both the bits of the operands are 1
- Example: **and $t0, $t1, $t2**

  $t1: 0000 0000 0000 0000 0000 1101 0000 0000
  $t2: 0000 0000 0000 0000 0011 1100 0000 0000
  $t0: 0000 0000 0000 0000 0000 1100 0000 0000

- AND can be used to create a mask. Force 0s into the positions that you are not interested; other bits will remain the same as the original.

Bitwise AND Instruction (2/2)

- Example: **andi $t0, $t1, 0xFFF**
  $t1: 0000 1001 1100 0011 0101 1101 1001 1100
  0xFFF: 0000 0000 0000 0000 0000 1111 1111 1111
  $t0:
- In the above example we are interested in the last 12 bits of the word in register **$t1**. So we use **0xFFF** as the mask.

Bitwise OR Instruction

- OR: bitwise operation that places a 1 in the result if either operand bit is 1
- Can be used to force certain bits to 1s
- Example: **ori $t0, $t1, 0xFFF**
  $t1: 0000 1001 1100 0011 0101 1101 1001 1100
  OxFFF: 0000 0000 0000 0000 0000 1111 1111 1111
  $t0:
- If both operands are registers then use **or**
  - **Example: or $t0, $t1, $t2**

Bitwise NOR Instruction

- Required operation is NOT; toggles the bits of an operand (1 with 0, 0 with 1)
- To maintain regularity (simplicity favors regularity), MIPS uses NOR instead of NOT
  - If one operand of NOR is 0, then it is equivalent to NOT:
    A NOR 0 = NOT(A OR 0) = NOT(A)
- Example: **nor $t0, $t1, $zero**

  $t1: 0000 1001 1100 0011 0101 1101 1001 1100

  $t0:
- There is no **nori** in MIPS – why?

MIPS Opcodes

- Learn by examples.
- Focus mainly on those simple ones, such as:

load
store
✓ add
✓ subtract
move register-register
✓ and
✓ shift
compare equal, compare not equal
branch
jump
call
return

MIPS

- The MIPS instruction set is typical of RISC architectures.
- MIPS is a load-store register architecture
  - 32 registers, each 32-bit (4-byte) long
  - Each word contains 32 bits (4 bytes)
  - Memory addresses are 32-bit long

MIPS Operands: Registers and Memory

Name	Examples	Comments
32 registers	$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at	Fast locations for data. In MIPS, data must be in registers to perform arithmetic.
2^{30} memory words	Mem[0], Mem[4], …, Mem[4294967292].	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

Memory Access Instructions

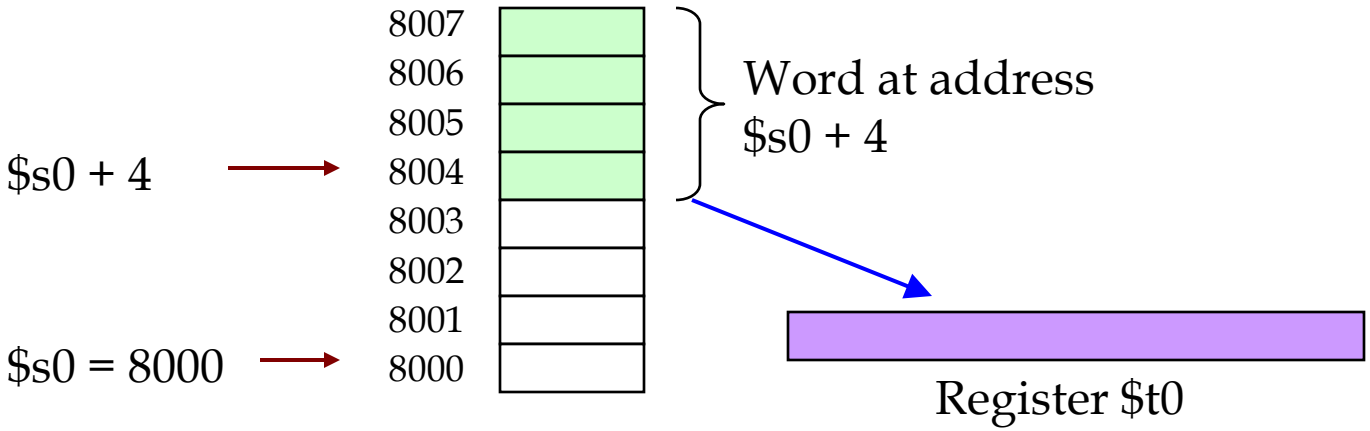
- Only **load** and **store** instructions can access memory data.
- Example: Each array element occupies a word.
  - C code: **A[7] = h + A[10];**
  - MIPS code:

```
lw      $t0, 40($s3)
add     $t0, $s2, $t0
sw      $t0, 28($s3)
```

  - Each array element occupies a word (4 bytes).
  - $s3 contains the **base address** (address of first element) of array A.
- Remember arithmetic operands (for add) are registers, not memory!

Load Word Instruction

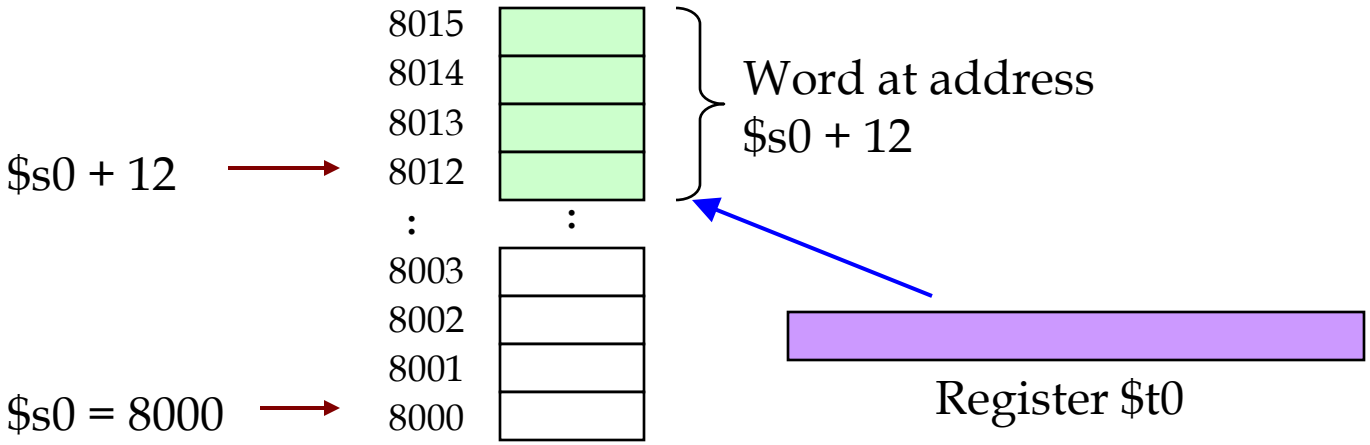
- Example: **lw $t0, 4($s0)**



- Load (read) the content from the memory word at address $s0 + 4 to the register $t0.

Store Word Instruction

- Example: **sw $t0, 12($s0)**



- Store (write) the content of register $t0 to the memory word at address $s0 + 12.

Other LOAD/STORE Instructions (1/2)

- Besides the load word (**lw**) and store word (**sw**) instructions, there are also the load byte (**lb**) and store byte (**sb**) instructions.
- Similar in format:

```
lb  $t1, 12($s3)
sb  $t2, 13($s3)
```

- Similar in working except that one byte, instead of one word, is loaded or stored.

Other LOAD/STORE Instructions (2/2)

- MIPS disallows loading/storing a word that crosses the word boundary. Use the unaligned load word (ulw) and unaligned store word (usw) instructions instead. These are pseudo-instructions.
- There are other instructions such as
  - lh and sh: load halfword and store halfword
  - lwl, lwr, swl, swr: load word left, load word right, store word left, store word right.
  - And others.

Arrays (1/2)

- Variable **h** is associated with register **$s2**. Array **A** has starting (base) address in register **$s3**
- C statement: **A[3] = h + A[1];**
- Transfer **A[1]** from memory to register

```
lw  $t0, 4($s3)  # offset = 1×4 = 4
```

- Perform addition; reuse temp register **$t0**

```
add $t0, $s2, $t0
```

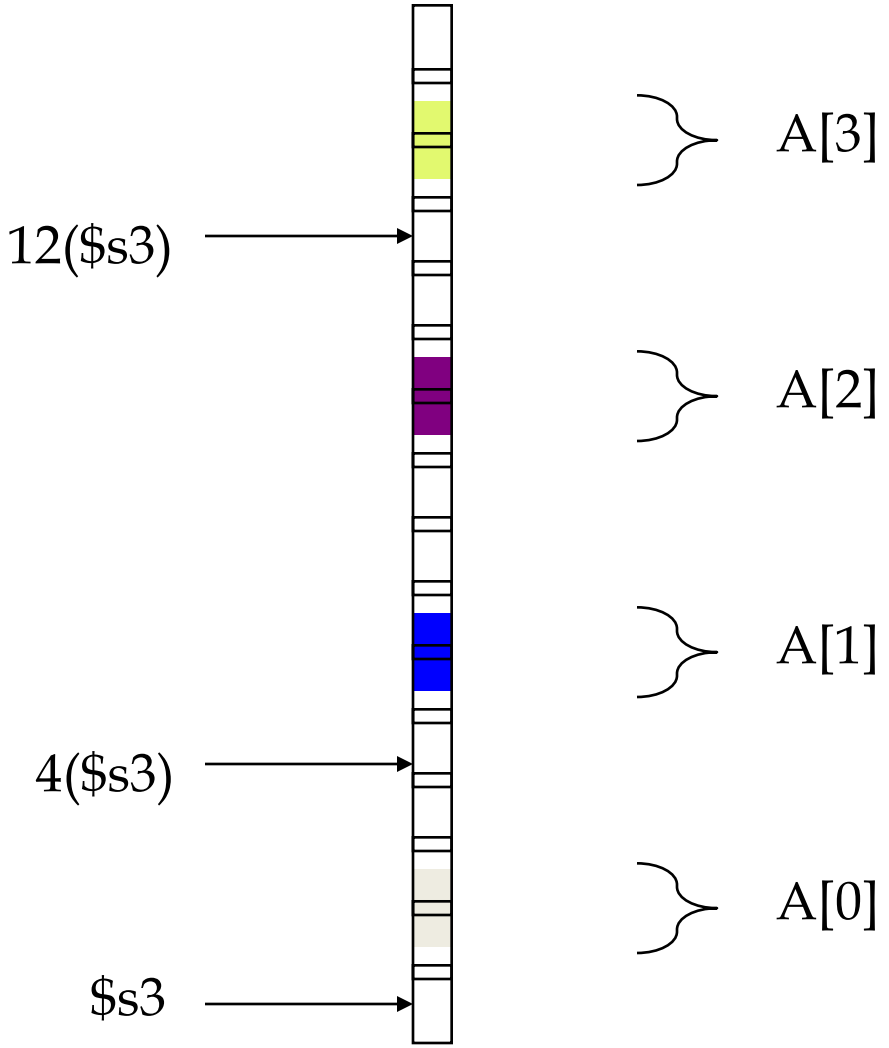
- Store the sum into **A[3]**

```
sw  $t0, 12($s3) # offset = 3×4 = 12
```

Arrays (2/2)

A[3] = h + A[1];

```
lw  $t0, 4($s3)
add $t0, $s2, $t0
sw  $t0, 12($s3)
```



ADDRESS versus VALUE

- Key concept: Registers do not have types
- A register can hold any 32-bit number. That number can be a value; but it can also be a memory address.
- If you write **add $t2, $t1, $t0** then **$t0** and **$t1** should contain data values.
- If you write **lw $t2,0($t0)** then **$t0** should contain a memory address.
- Do not mix these things up!

BYTE versus WORD

- Impt: Consecutive word addresses in machines with byte-addressing do not differ by 1.
- Many an assembly language programmer has toiled over errors made by assumption that the address of the next word can be found by incrementing the address in a register by 1 instead of by the word size in bytes.
- For both **lw** and **sw** the sum of base address and offset must be multiple of 4.

Example

```
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
```

READING ASSIGNMENT

- Instructions: Language of the Computer
  - Read up COD Chapter 2, pages 52-57. (3rd edition)
  - Read up COD Chapter 2, pages 80-85. (4th edition)

Constants

- Small constants are used quite frequently (50% of operands). Examples:

```
A = A + 5;
B = B + 1;
C = C – 18;
```

- Solution:
  - Put 'typical constants' in memory and load them.
  - Create hard-wired registers (like $zero) for constants like one.
  - Embed them into the instruction code.
  - For MIPS instructions, we use the 'immediate operand' version:

```
addi    $29, $29, 4
slti    $8, $18, 10
andi    $29, $29, 6
ori     $29, $29, 4
```

Large Constants (1/3)

- Recall bit-wise operations: **or**, **ori**, **and**, **andi**.
  - Examples:

1010 or 1001 → 1011

	1010
or	1001
	1011

1010 and 1001 → 1000

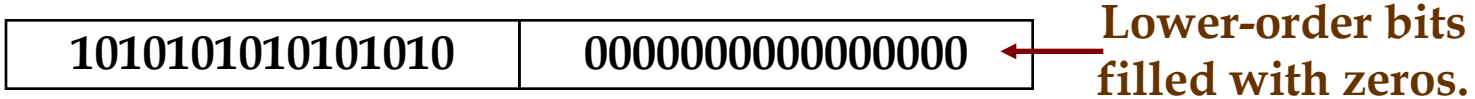
	1010
and	1001
	1000

- Question: How to load a 32-bit constant into a register?
  - Example: 1010101010101010 1111000011110000

Large Constants (2/3)

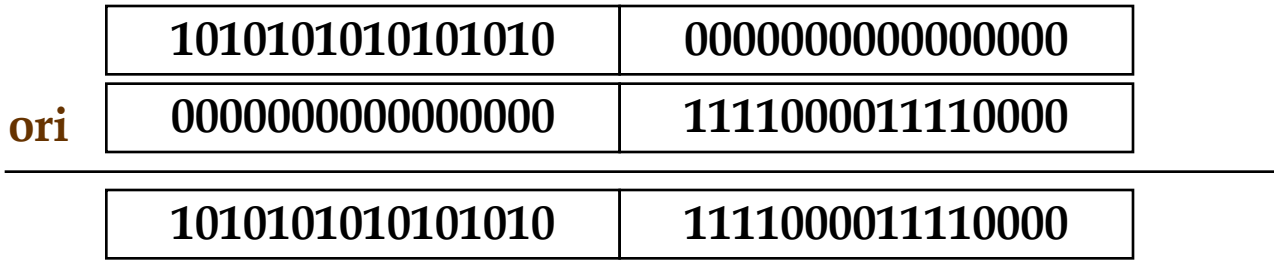
- Need to use two instructions: "load upper immediate" (**lui**) and "or immediate" (**ori**):

```
lui     $t0, 1010101010101010
```



- Then to get the lower-order bits correct:

```
ori     $t0, $t0, 1111000011110000
```



- Question: Why do we use **ori** instead of **addi**?

Large Constants (3/3)

- The above are just illustration of the concept. In the actual instruction, the value is entered as a decimal value or an hexadecimal value (prefixed by 0x).
- Example:

```
lui  $t0, 43690    # 43690_10 = 1010101010101010_2
```

or

```
lui  $t0, 0xAAAA   # AAAA_16 = 1010101010101010_2
```

- Example: To add **0xABABCDCD** to **$t0**:

```
lui  $at, 0xABAB
ori  $at, $at, 0xCDCD
add  $t0, $t0, $at
```

$at is a special register used by compiler to form large constants

READING ASSIGNMENT

- Instructions: Language of the Computer
  - Read up COD Chapter 2, pages 95-96. (3rd edition)
  - Read up COD Chapter 2, pages 128-129. (4th edition)

Making Decisions (1/2)

- So far all the instructions only manipulate data… we've built a calculator
- A computer needs the ability to make decisions
- High-level language decisions
  - **if** and **goto** statements
- MIPS decision making instructions are similar to **if** statement with a **goto**
- **goto** is discouraged in high-level languages but necessary in assembly ☺

Making Decisions (2/2)

- Decision-making instructions
  - Alter the control flow of the program
  - Change the next instruction to be executed
- Two type of decision-making statements
  - Conditional (branch)

    bne $t0, $t1, label

    beq $t0, $t1, label
  - Unconditional (jump)

    j label

Conditional Branch

- Processor follows the branch conditionally
- **beq $r1,$r2, L1**
  - go to statement labeled **L1** if the value in register **$r1** equals the value in register **$r2**
  - **beq** is "branch if equal"
  - C code: **if (a == b) goto L1**
- **bne $r1, $r2, L1**
  - go to statement labeled **L1** if the value in register **$r1** does not equal the value in register **$r2**
  - **bne** is "branch if not equal"
  - C code: **if (a != b) goto L1**

Unconditional Jump

- Processor always follows the branch
- **j L1**
  - Jump to label **L1** unconditionally
  - C code: **goto L1**
- Technically equivalent to such statement

  beq $s0, $s0, L1

IF Statement (1/2)

- C statement: **if (i == j) f = g + h;**
- Mapping **f:$s0; g:$s1; h:$s2; i:$s3; j:$s4**
- MIPS statements:

```
    bne $s3, $s4, Exit  # if (i!=j) Exit
    add $s0, $s1, $s2   # f = g + h
Exit:
```

- Why use **bne** instead of **beq**? For efficiency.

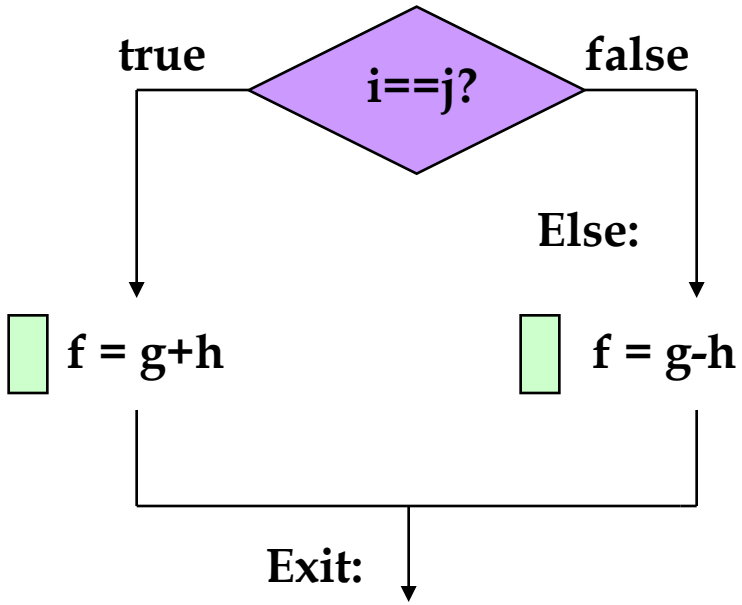
```
    beq $s3, $s4, L1    # if (i==j) goto L1
    j Exit              # goto Exit
L1: add $s0, $s1, $s2   # f = g + h
Exit:
```

IF Statement (2/2)

- C statement:
  if (i == j) f = g + h;
  else f = g - h;
- Mapping **f:$s0; g:$s1; h:$s2; i:$s3; j:$s4**
- MIPS statements:

```
        bne $s3, $s4, Else
        add $s0, $s1, $s2
        j   Exit
Else: sub $s0, $s1, $s2
Exit:
```

Q: Rewrite using **beq**



Try It Yourself #1

- Mapping **f:$s0; i:$s1; j:$s2**
- MIPS assembly

```
    beq $s1, $s2, Exit
    add $s0, $zero, $zero
Exit:
```

- What is the corresponding high-level statement?

Loops (1/2)

- C statement: **while (j == k) i = i+1;**
- Rewrite code with if and goto statements

```
Loop: if (j != k) Exit;
      i = i+1;
      goto Loop;
Exit:
```

- Key concept: The key to decision making is conditional branches. Any form of loop can be written in assembly with the help of conditional branches and jumps.

Loops (2/2)

- Code:

```
Loop: if (j != k) Exit;
      i = i+1;
      goto Loop;
Exit:
```

- Write MIPS statement using mapping **i:$s3; j:$s4; k:$s5**

Try It Yourself #2

- Write the following loop statement in MIPS

```
for (i=0; i<10; i++) a = a + 5;
```

Mapping **`i:$s0;a:$s2`**

Inequalities (1/2)

- We have **beq** and **bne**, what about branch-if-less-than? We do not have a **blt** instruction.
- Use **slt** (set on less than) or **slti**.

```
slt $t0, $s1, $s2
```

=

```
if ($s1 < $s2)
        $t0 = 1;
else
        $t0 = 0;
```

Inequalities (2/2)

- To build a **"blt $s1, $s2, L"** instruction:

```
slt $t0, $s1, $s2
bne $t0, $zero, L
```

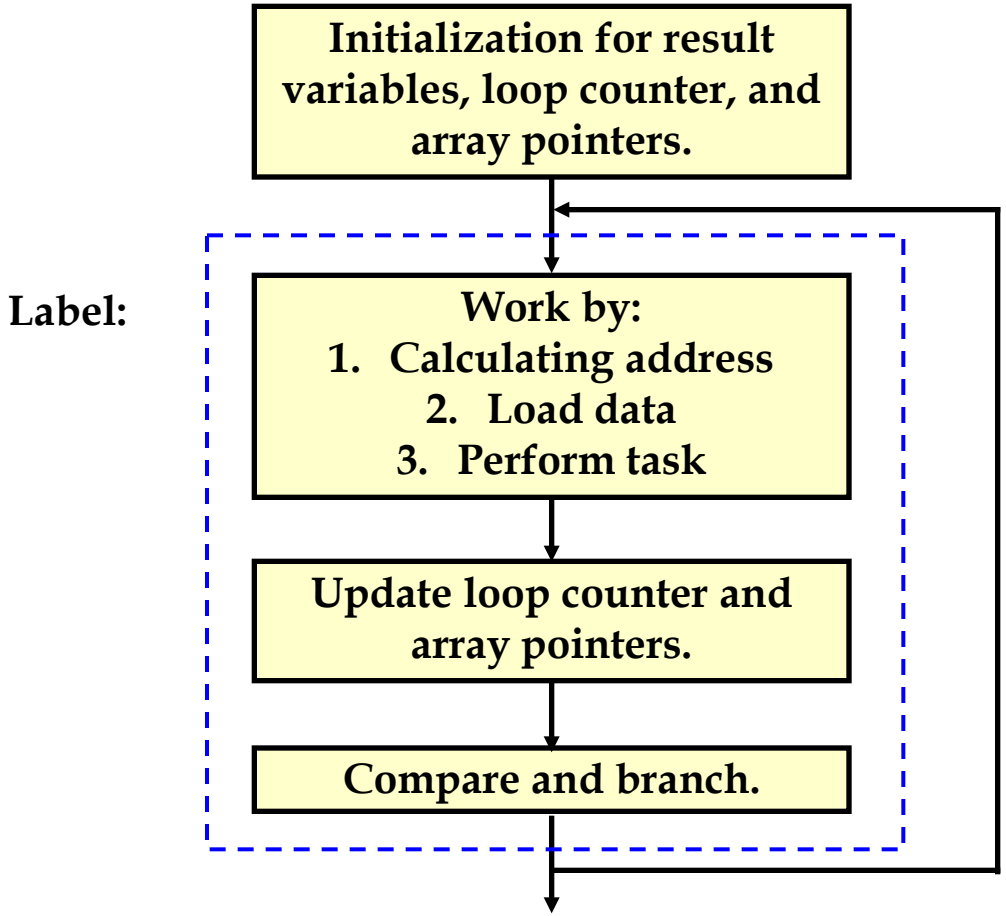
C code: **if (a < b) goto L;**

READING ASSIGNMENT

- Instructions: Language of the Computer
  - Section 2.6 Instructions for Making Decisions. (3rd edition)
  - Section 2.7 Instructions for Making Decisions. (4th edition)

Array and Loop

- Typical example of accessing array elements in a loop:



Array and Loop: Example (1/3)

- Given a word array A with 40 elements A[0], A[1], …, A[39], with the starting array address stored in $t0. Count the number of array elements with value zero, and store the result in $t8.

Answer 1:

```
        addi   $t8, $zero, 0      # initialize counter to zero
        addi   $t1, $zero, 0      # $t1 as index, init. to point to 1st element
L1:     add    $t2, $t0, $t1      # address of current element
        lw     $t3, 0($t2)        # load current element from memory
        bne    $zero, $t3, L2     # check if data is zero, if not, skip update
        addi   $t8, $t8, 1        # increment counter
L2:     addi   $t1, $t1, 4        # point to next element
```

How to compare and loop back?

```
bne    $t1, 160, L1 ? No immediate compare operand!
```

Array and Loop: Example (2/3)

Answer 2:

```
      addi  $t8, $zero, 0      # initialize counter to zero
      addi  $t1, $zero, 156    # $t1 as index, init. to pt to last element
L1:   add   $t2, $t0, $t1      # address of current element
      lw    $t3, 0($t2)        # load current element from memory
      bne   $zero, $t3, L2     # check if data is zero, if not, skip update
      addi  $t8, $t8, 1        # increment counter
L2:   addi  $t1, $t1, –4       # point to previous element
      bne   $t1, $zero, L1
```

How about the iteration for the first element?

Array and Loop: Example (3/3)

Answer 3:

```
        addi   $t8, $zero, 0      # initialize counter to zero
        addi   $t1, $zero, ???    # $t1 as index, init. to point pass
                                  # last element
L1:     add    $t2, $t0, $t1      # address of current element
        lw     $t3, ???($t2)      # load preceding element from memory
        bne    $zero, $t3, L2     # check if data is zero, if not, skip update
        addi   $t8, $t8, 1        # increment counter
L2:     addi   $t1, $t1, –4       # point to previous element
        bne    $t1, $zero, L1
```

Try It Yourself #3

- Given the following MIPS code:

```
        addi $t1, $zero, 10
        add  $t1, $t1, $t1
        addi $t2, $zero, 10
Loop1:  addi $t2, $t2, 10
        addi $t1, $t1, -1
        beq  $t1, $zero, Loop1
```

i. How many instructions are executed?

(a) 6 (b) 30 (c) 33 (d) 36 (e) None of the above

ii. What is the final value in $t2?

(a) 10 (b) 20 (c) 300 (d) 310 (e) None of the above

Try It Yourself #4

- Given the following MIPS code:

```
        add  $t0, $zero, $zero
        add  $t1, $t0, $t0
        addi $t2, $t1, 4
Loop1:  add  $t1, $t1, $t0
        addi $t0, $t0, 1
        bne  $t2, $t0, Loop1
```

i. How many instructions are executed?
(a) 6 (b) 12 (c) 15 (d) 18 (e) None of the above

ii. What is the final value in $t1?
(a) 0 (b) 4 (c) 6 (d) 10 (e) None of the above

iii. Assume CPIs for **add** and **addi** are 1, and **bne** is 4, what is the average CPI for the code?
(a) 1.4 (b) 1.8 (c) 2.0 (d) 2.2 (e) None of the above

Try It Yourself #5 (1/2)

- Given a word array of elements in memory with the starting address stored in $t0, the following MIPS code is executed:

```
        addi $t1, $t0, 10
        add  $t2, $zero, $zero
Loop1:  ulw  $t3, 0($t1)
        add  $t2, $t2, $t3
        addi $t1, $t1, -1
        bne  $t1, $t0, Loop1
```

i. How many times is the **bne** instruction executed?
(a) 1 (b) 3 (c) 9 (d) 10 (e) 11

ii. How many instructions are executed?
(a) 6 (b) 12 (c) 41 (d) 42 (e) 46

Try It Yourself #5 (2/2)

- Given a word array of elements in memory with starting address stored in $t0, the following code is executed:

```
        addi $t1, $t0, 10
        add  $t2, $zero, $zero
Loop1:  ulw  $t3, 0($t1)
        add  $t2, $t2, $t3
        addi $t1, $t1, -1
        bne  $t1, $t0, Loop1
```

iii. How many times does the **bne** instruction actually branch to the label **Loop1**?

(a) 1 (b) 8 (c) 9 (d) 10 (e) 11

iv. How many unique bytes of memory (excluding registers) are read?

(a) 4 (b) 10 (c) 11 (d) 13 (e) 40

Concepts Of Focus

- Basic MIPS programming
  - Arithmetic: among registers only
  - Memory accesses: load/store
  - Control flow: branch and jump
  - Accessing array elements
  - Handling of large constants
- Things we are not going to cover
  - Support for procedures
  - Linkers, loaders, memory layout
  - Stacks, frames, recursion
  - Interrupts and exceptions
  - System calls

MIPS Simulator

- Try out PCSpim (MIPS simulator)
  - ftp://ftp.cs.wisc.edu/pub/spim/pcspim.exe
    or
  - http://spimsimulator.sourceforge.net/