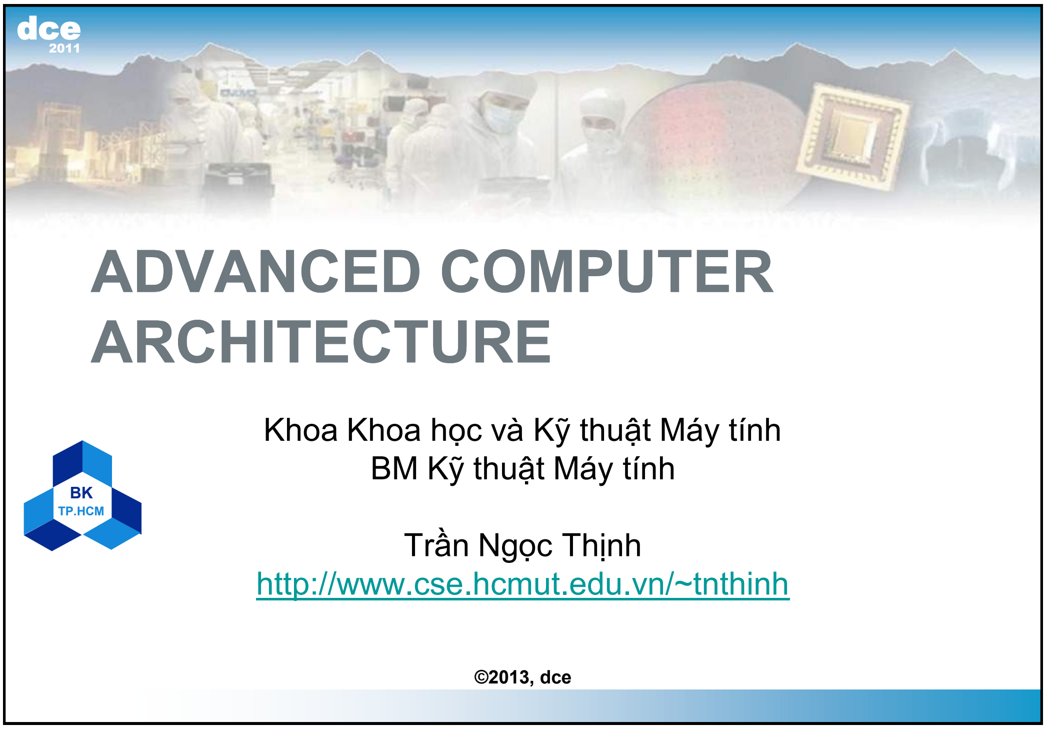
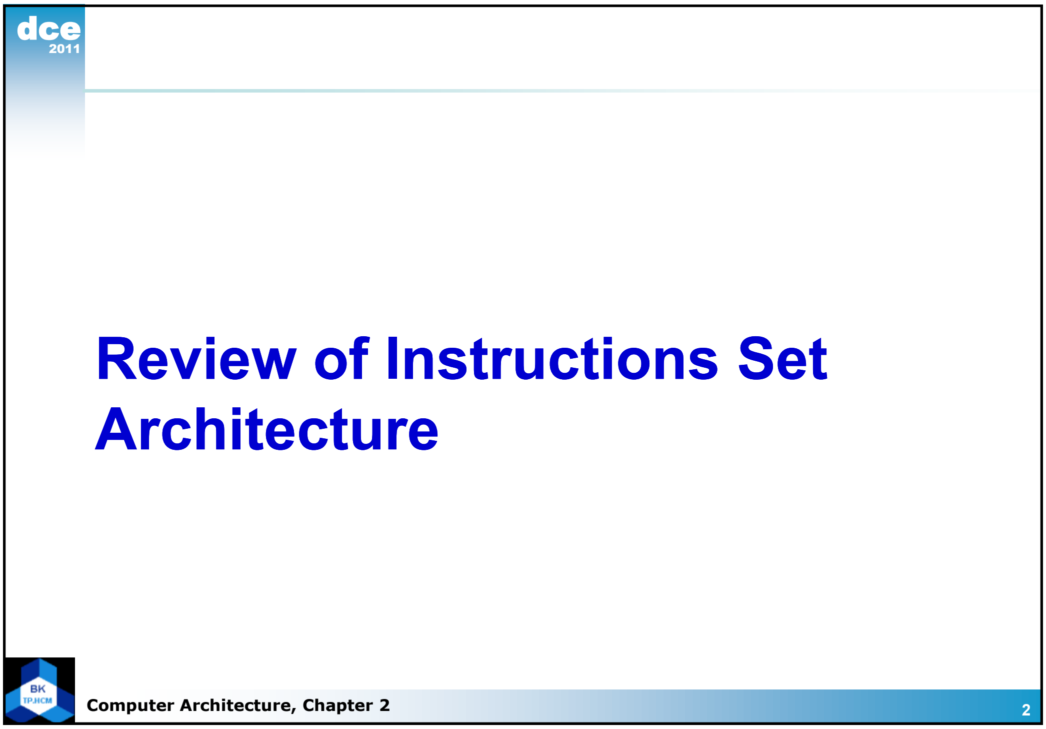


dce 2011

ADVANCED COMPUTER ARCHITECTURE

Khoa Khoa học và Kỹ thuật Máy tính
BM Kỹ thuật Máy tính

Trần Ngọc Thịnh
http://www.cse.hcmut.edu.vn/~tnthinh



dce 2011

Review of Instructions Set Architecture

Outline

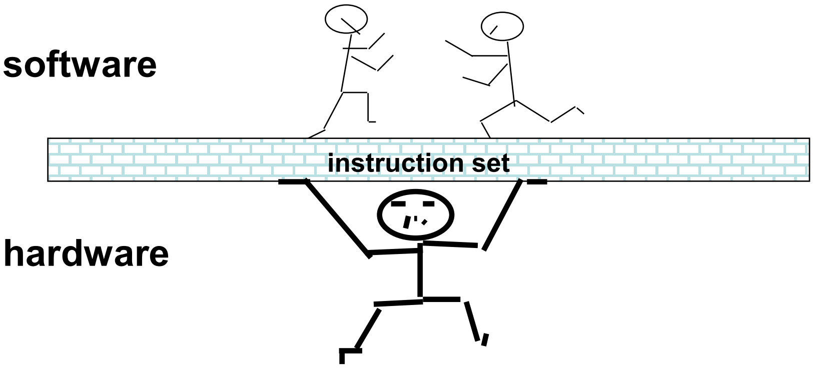
- Instruction structure
- ISA styles
- Addressing modes
- Analysis on instruction set
- Case study: MIPS

Machine Instruction

Computer can only understand binary values

The operation of a computer is defined by predefined binary values called *Instruction*

The Instruction Set



software

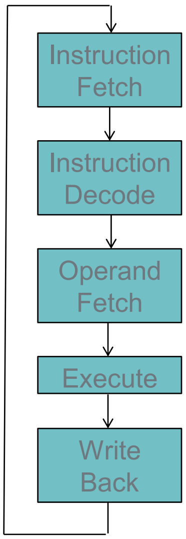
instruction set

hardware

Instruction set: set of all instructions a processor can perform

Interface between software and hardware

Instruction execution cycle



Instruction Fetch

Obtain the instruction

Instruction Decode

Determine action to be perform

Operand Fetch

Obtain the operand data

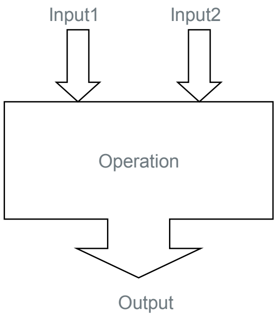
Execute

Compute the result (update status)

Write Back

Store the result

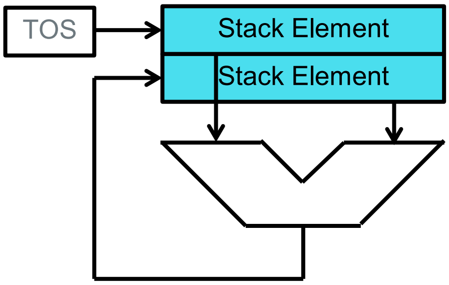
ISA Styles



ISA Styles?

- Stack
- Accumulator
- Register memory/ Memory memory
- Register register/load store

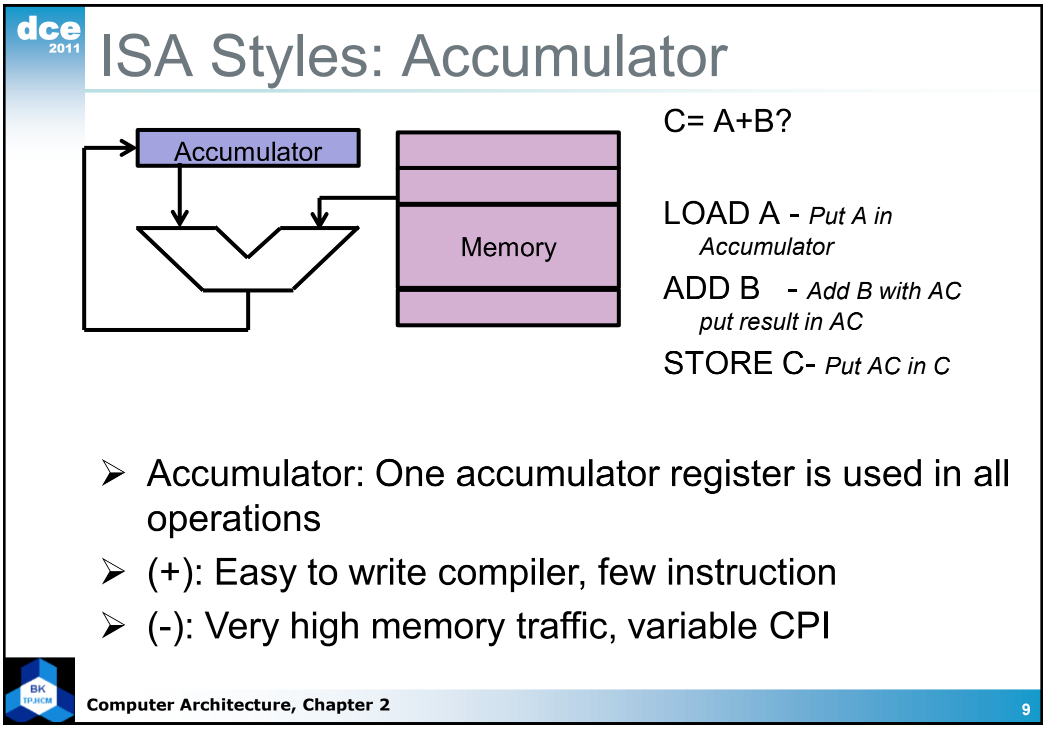
ISA Styles: Stack



C= A+B?

PUSH A
PUSH B
ADD
POP C

- Stack: The operands is on top of stack. The result is push back to the stack
- (+): Code density, simple hardware
- (-): Low parallelism, stack bottle-neck



ISA Styles: Accumulator

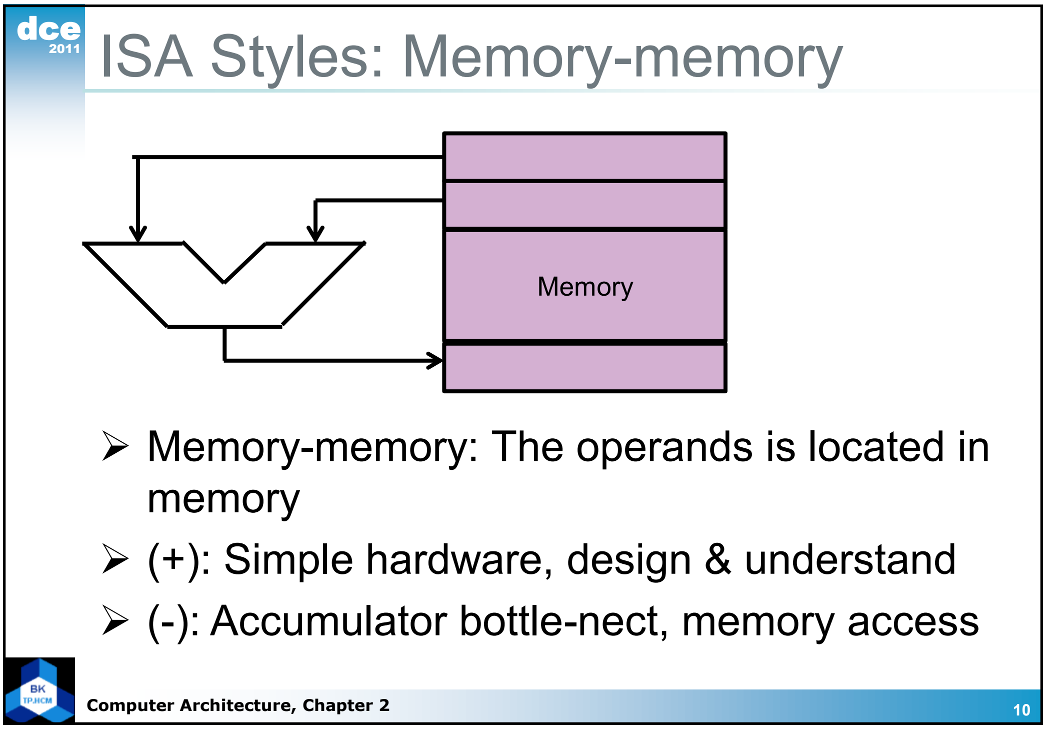
C= A+B?

LOAD A - *Put A in Accumulator*

ADD B - *Add B with AC put result in AC*

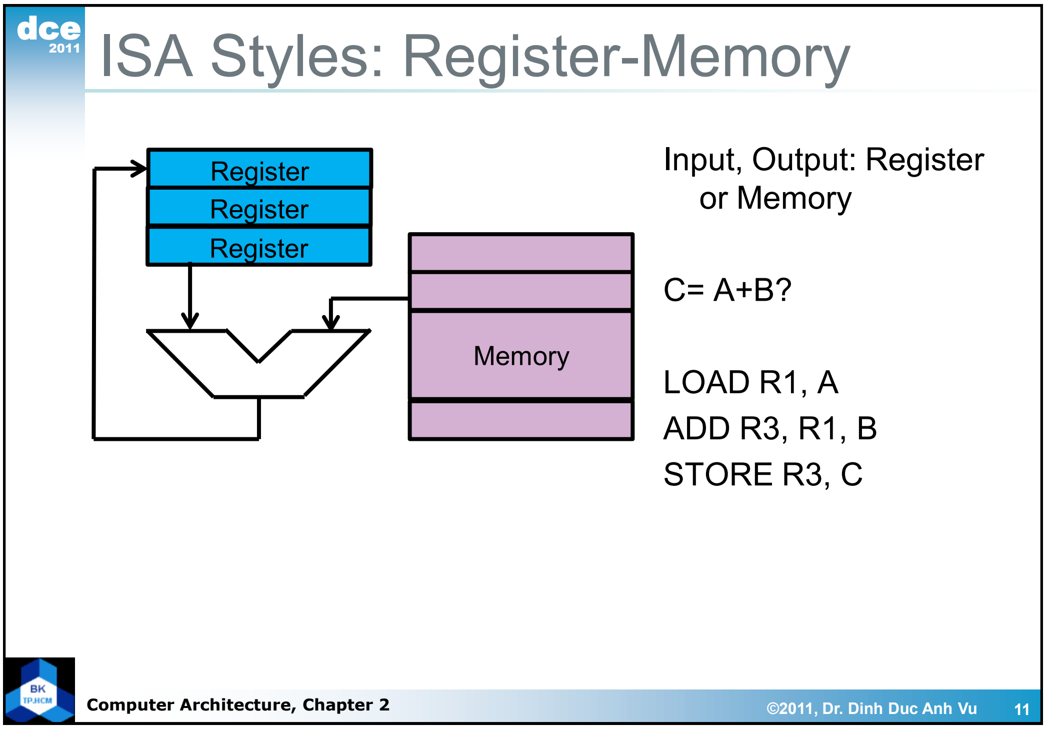
STORE C- *Put AC in C*

- Accumulator: One accumulator register is used in all operations
- (+): Easy to write compiler, few instruction
- (-): Very high memory traffic, variable CPI



ISA Styles: Memory-memory

- Memory-memory: The operands is located in memory
- (+): Simple hardware, design & understand
- (-): Accumulator bottle-nect, memory access

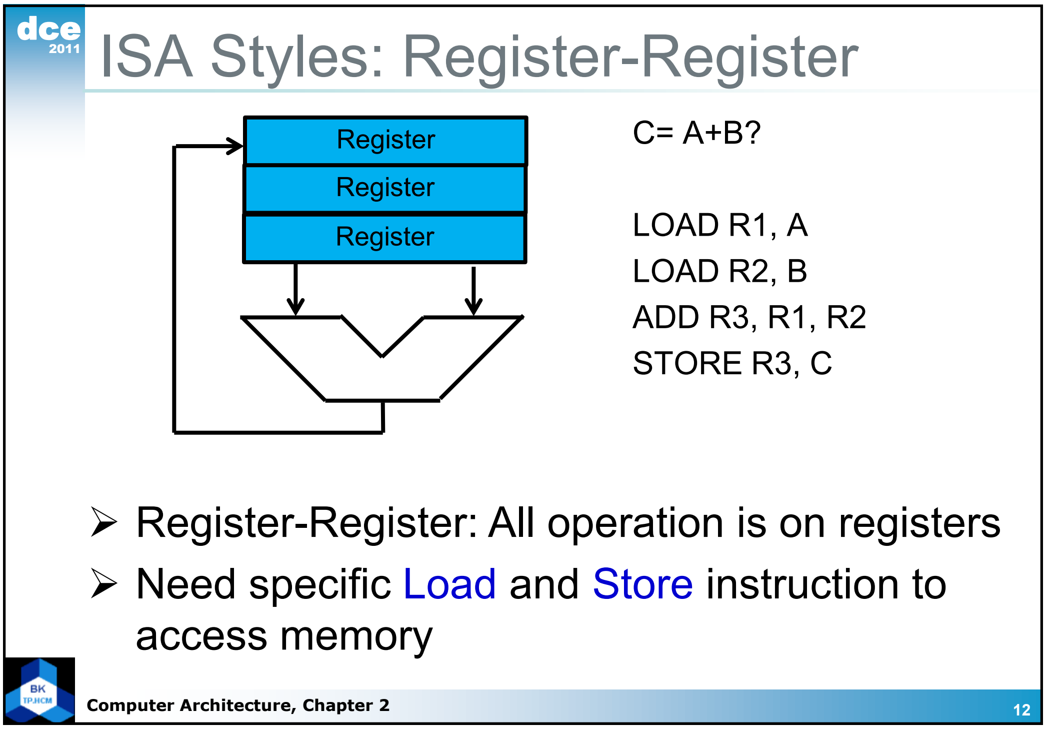


ISA Styles: Register-Memory

Input, Output: Register or Memory

C= A+B?

LOAD R1, A
ADD R3, R1, B
STORE R3, C

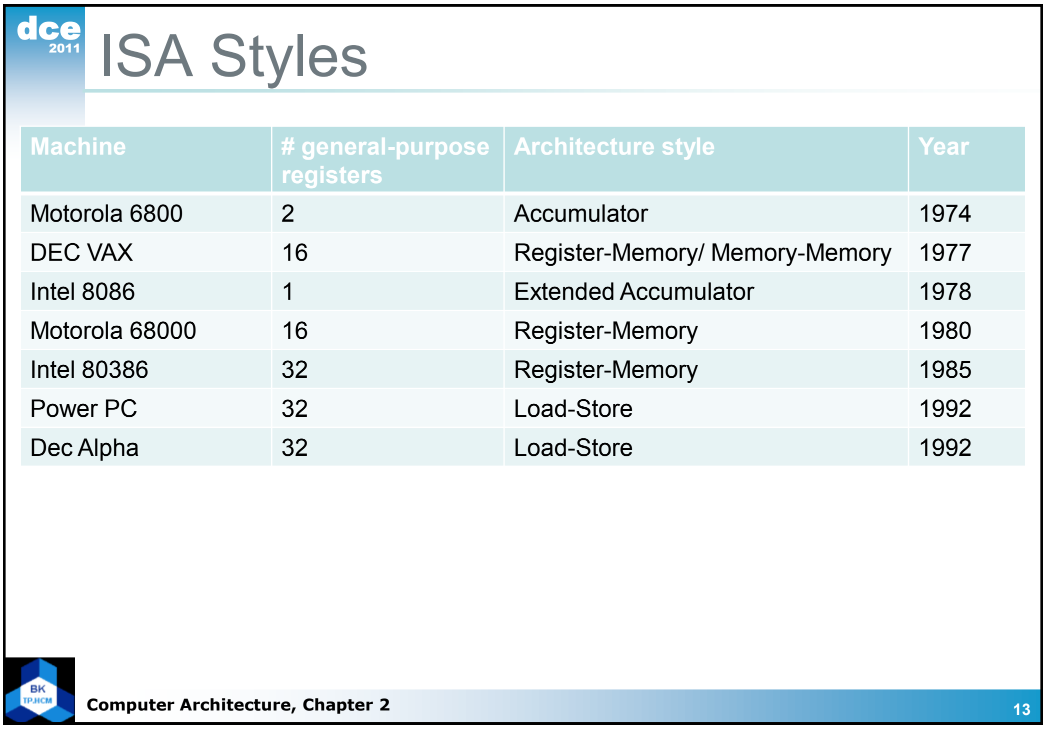


ISA Styles: Register-Register

C= A+B?

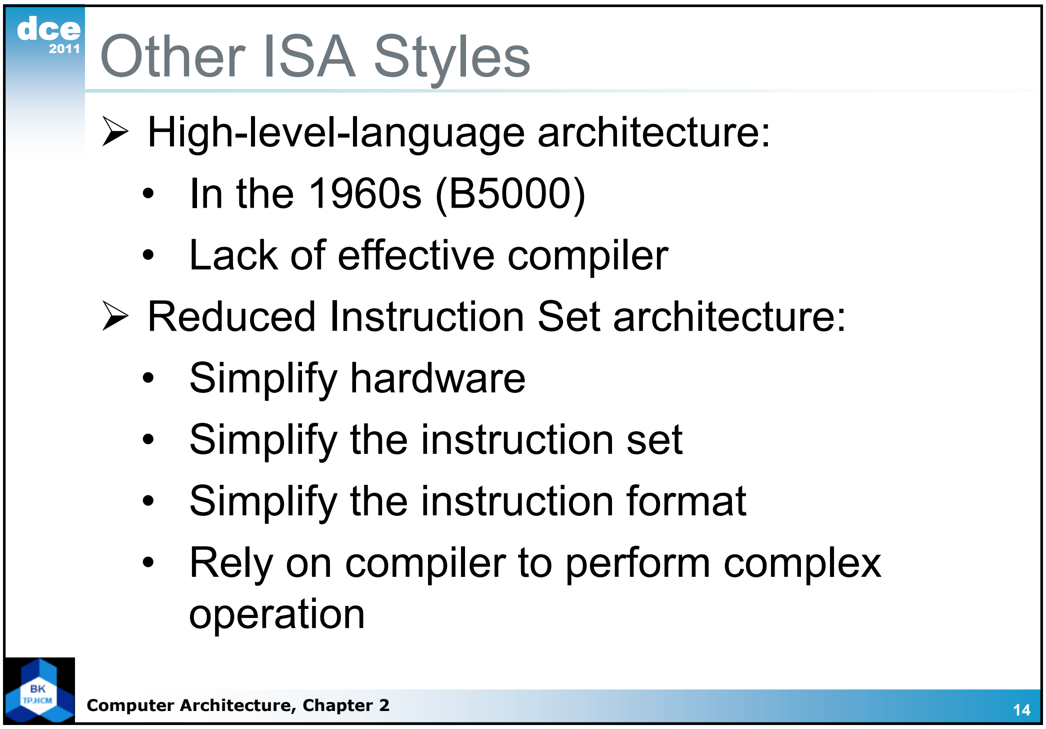
LOAD R1, A
LOAD R2, B
ADD R3, R1, R2
STORE R3, C

- Register-Register: All operation is on registers
- Need specific Load and Store instruction to access memory



ISA Styles

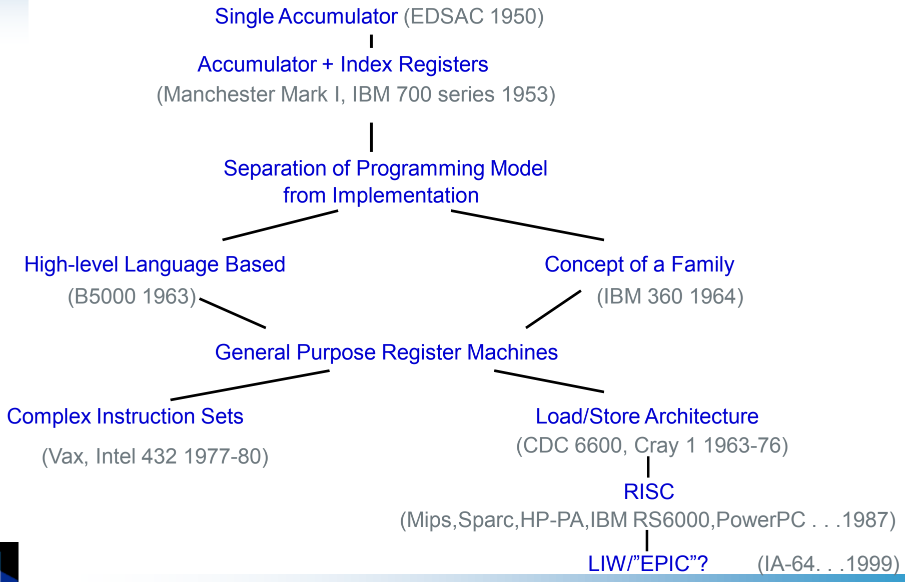
Machine	# general-purpose registers	Architecture style	Year
Motorola 6800	2	Accumulator	1974
DEC VAX	16	Register-Memory/ Memory-Memory	1977
Intel 8086	1	Extended Accumulator	1978
Motorola 68000	16	Register-Memory	1980
Intel 80386	32	Register-Memory	1985
Power PC	32	Load-Store	1992
Dec Alpha	32	Load-Store	1992



Other ISA Styles

- ➢ High-level-language architecture:
  - In the 1960s (B5000)
  - Lack of effective compiler
- ➢ Reduced Instruction Set architecture:
  - Simplify hardware
  - Simplify the instruction set
  - Simplify the instruction format
  - Rely on compiler to perform complex operation

Evolution of Instruction Sets



Single Accumulator (EDSAC 1950)

Accumulator + Index Registers
(Manchester Mark I, IBM 700 series 1953)

Separation of Programming Model
from Implementation

High-level Language Based
(B5000 1963)

Concept of a Family
(IBM 360 1964)

General Purpose Register Machines

Complex Instruction Sets
(Vax, Intel 432 1977-80)

Load/Store Architecture
(CDC 6600, Cray 1 1963-76)

RISC
(Mips,Sparc,HP-PA,IBM RS6000,PowerPC . . .1987)

LIW/"EPIC"? (IA-64. . .1999)

Instruction set design

- The design of an Instruction Set is critical to the operation of a computer system.
- Including many aspects
  - Operation repertoire
  - Addressing modes
  - Data types
  - Instruction format
  - Registers

Simple format

Opcode	Operand reference	Operand reference

Operation Code: the operation to be performed by the processor

Source Operand Reference: Input of the operation. One or more source operands can be involved

Result Operand Reference: Result of the operation

Instruction Types

Can be classified into 4 types:

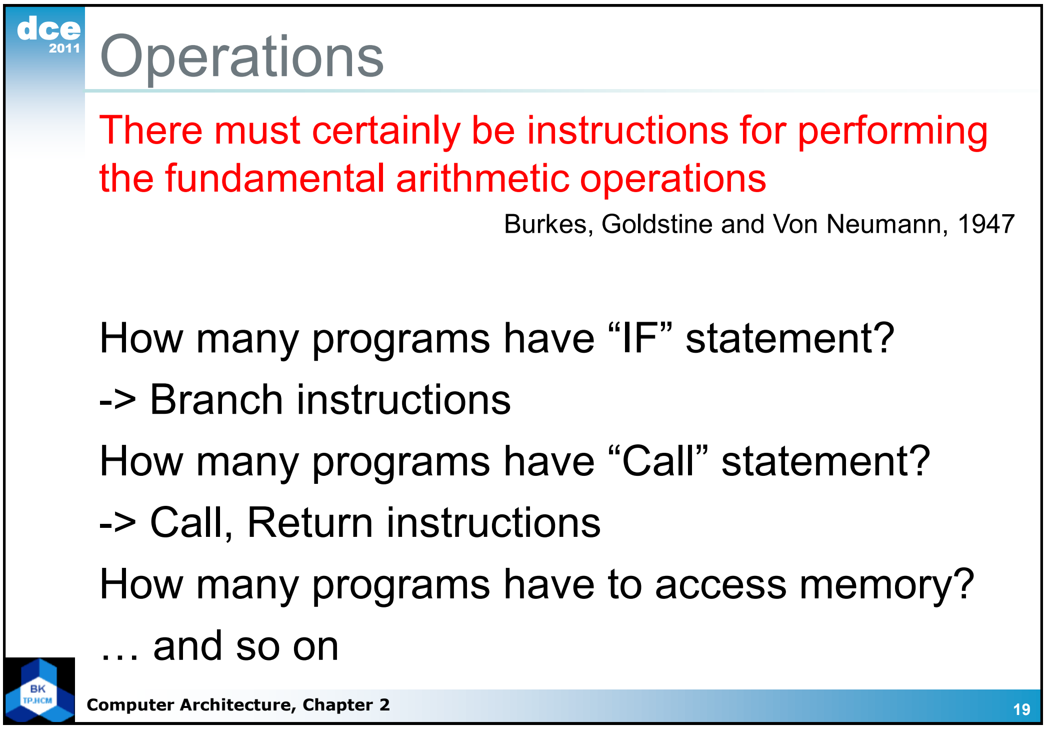
- Data processing: Arithmetic, Logic

  Ex: ADD, SUB, AND, OR, …
- Data storage: Move data from/to memory

  Ex: LD, ST
- Data movement: Register and register/IO

  Ex: MOV
- Control: Test and branch

  Ex: JMP, CMP



Operations

There must certainly be instructions for performing the fundamental arithmetic operations

Burkes, Goldstine and Von Neumann, 1947

How many programs have “IF” statement?

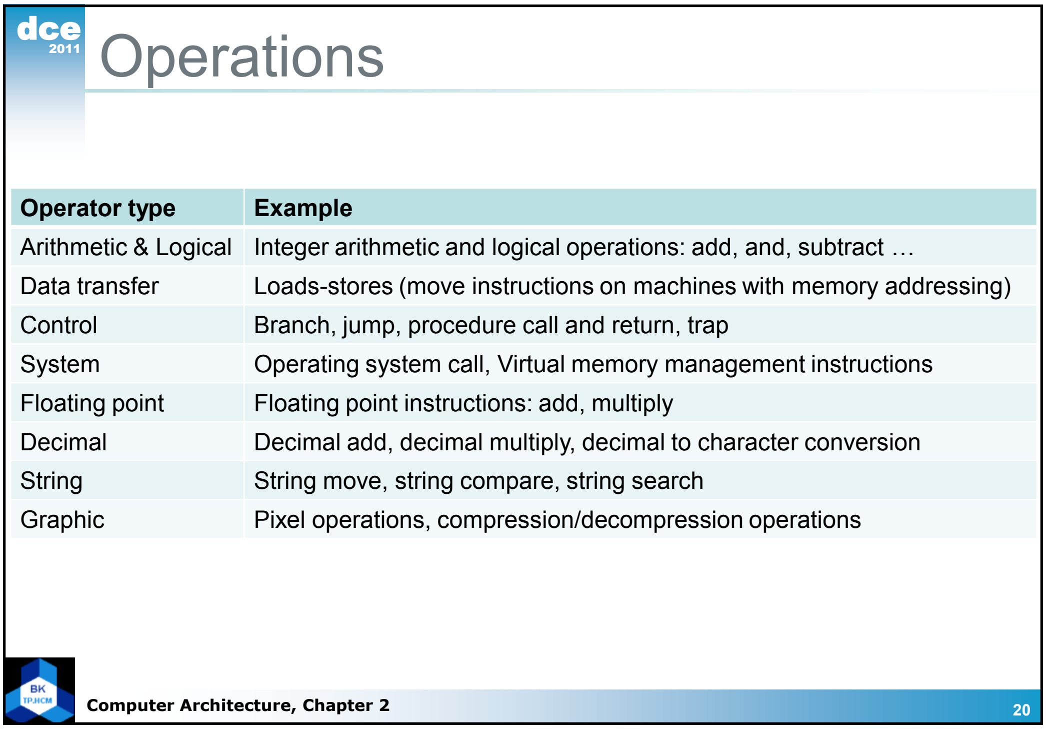
-> Branch instructions

How many programs have “Call” statement?

-> Call, Return instructions

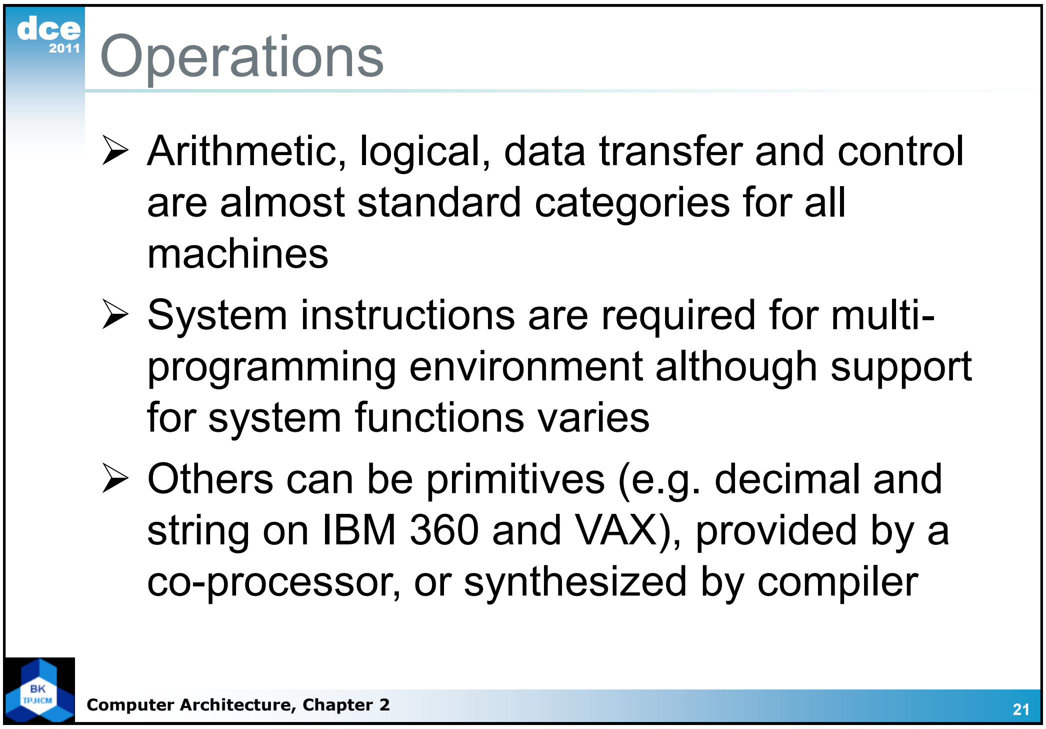
How many programs have to access memory?

… and so on



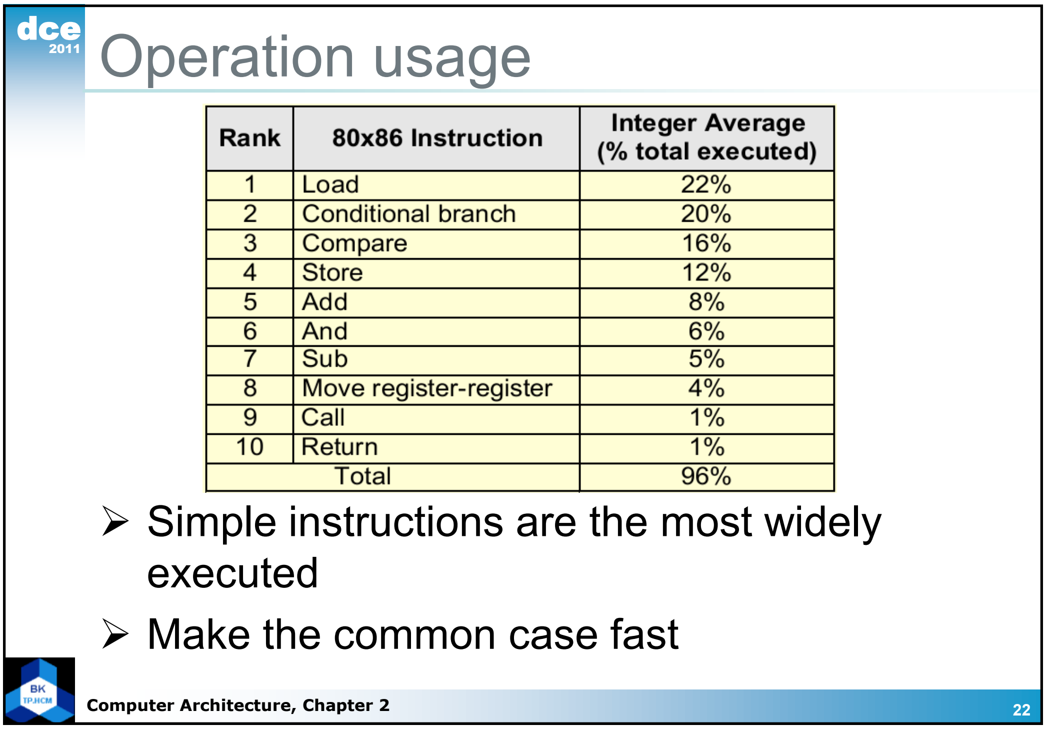
Operations

Operator type	Example
Arithmetic & Logical	Integer arithmetic and logical operations: add, and, subtract …
Data transfer	Loads-stores (move instructions on machines with memory addressing)
Control	Branch, jump, procedure call and return, trap
System	Operating system call, Virtual memory management instructions
Floating point	Floating point instructions: add, multiply
Decimal	Decimal add, decimal multiply, decimal to character conversion
String	String move, string compare, string search
Graphic	Pixel operations, compression/decompression operations



Operations

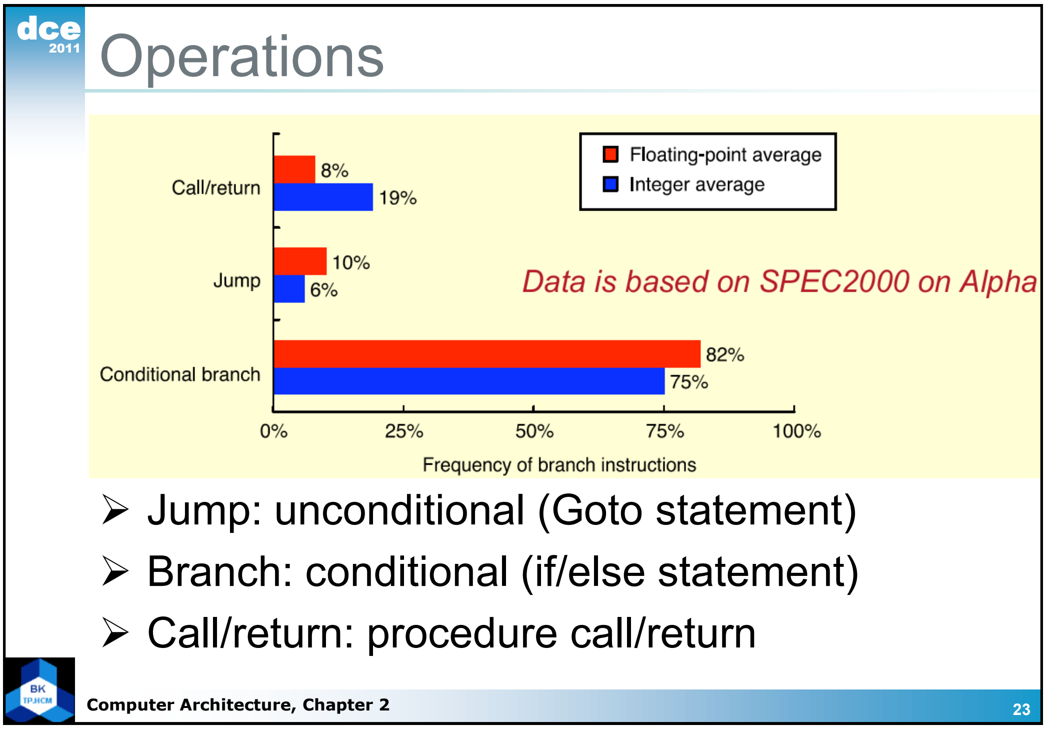
- ➢ Arithmetic, logical, data transfer and control are almost standard categories for all machines
- ➢ System instructions are required for multi-programming environment although support for system functions varies
- ➢ Others can be primitives (e.g. decimal and string on IBM 360 and VAX), provided by a co-processor, or synthesized by compiler



Operation usage

Rank	80x86 Instruction	Integer Average (% total executed)
1	Load	22%
2	Conditional branch	20%
3	Compare	16%
4	Store	12%
5	Add	8%
6	And	6%
7	Sub	5%
8	Move register-register	4%
9	Call	1%
10	Return	1%
Total		96%

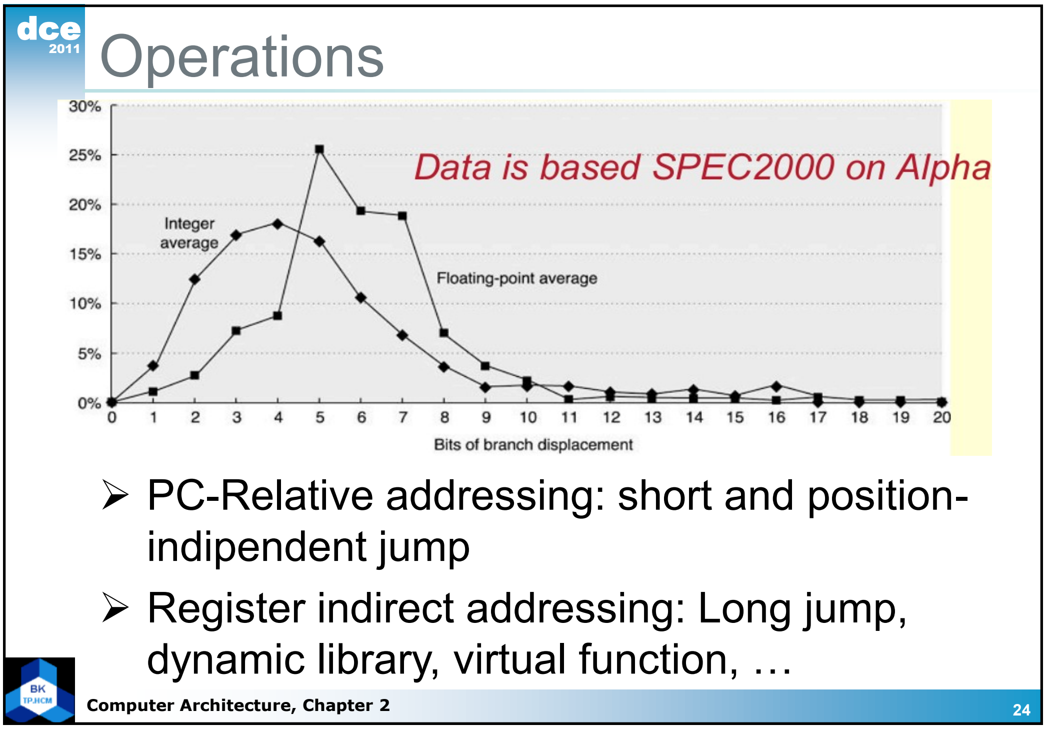
- ➢ Simple instructions are the most widely executed
- ➢ Make the common case fast



Operations

Data is based on SPEC2000 on Alpha

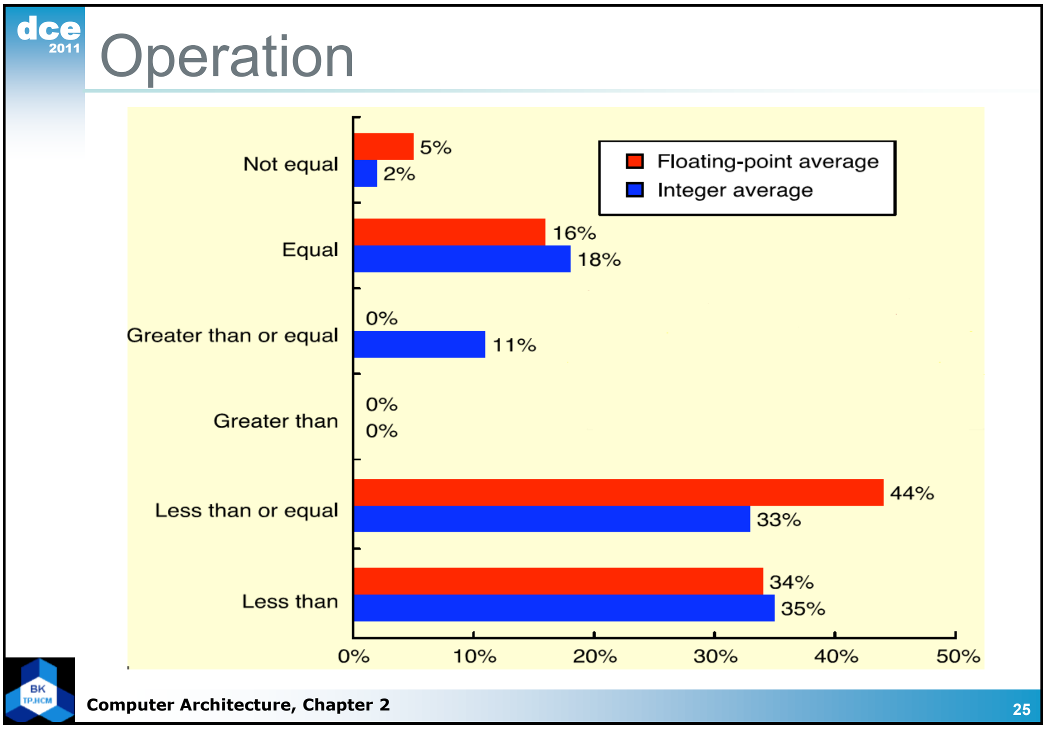
- Jump: unconditional (Goto statement)
- Branch: conditional (if/else statement)
- Call/return: procedure call/return



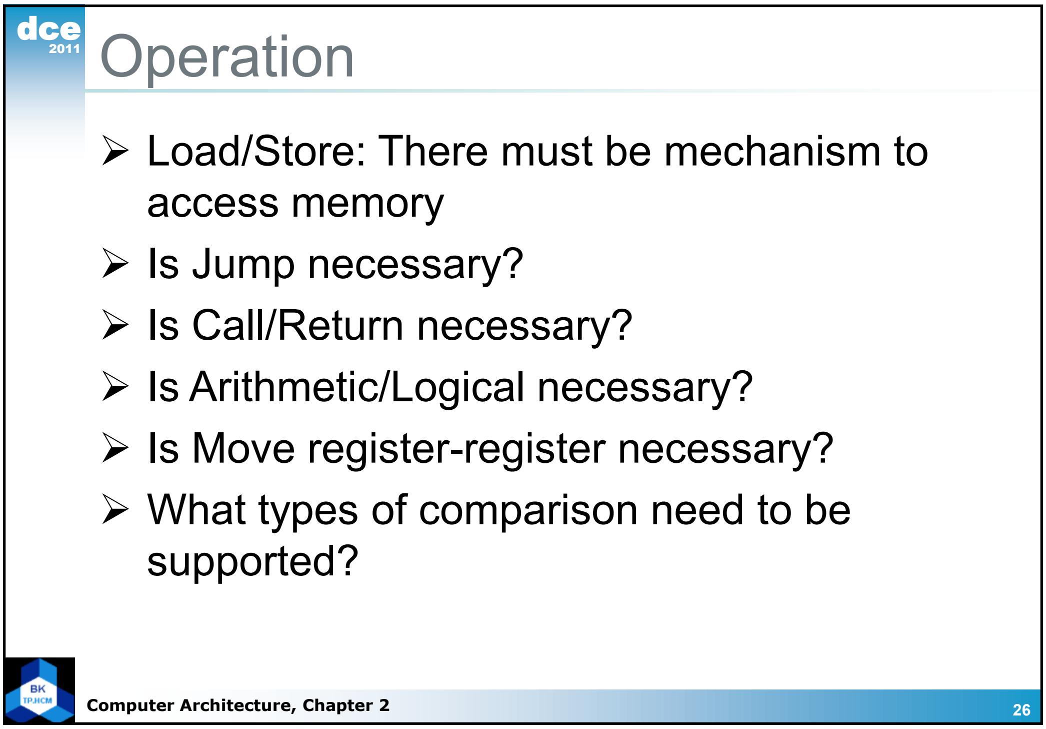
Operations

Data is based SPEC2000 on Alpha

- PC-Relative addressing: short and position-indipendent jump
- Register indirect addressing: Long jump, dynamic library, virtual function, …

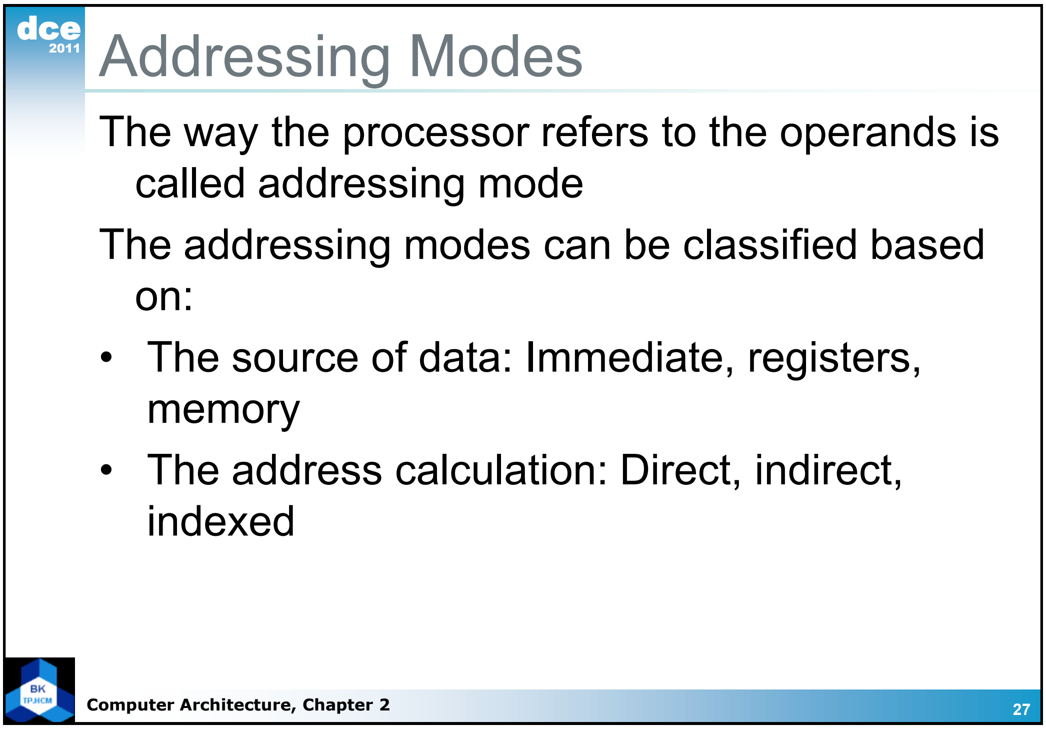


Operation



Operation

- Load/Store: There must be mechanism to access memory
- Is Jump necessary?
- Is Call/Return necessary?
- Is Arithmetic/Logical necessary?
- Is Move register-register necessary?
- What types of comparison need to be supported?

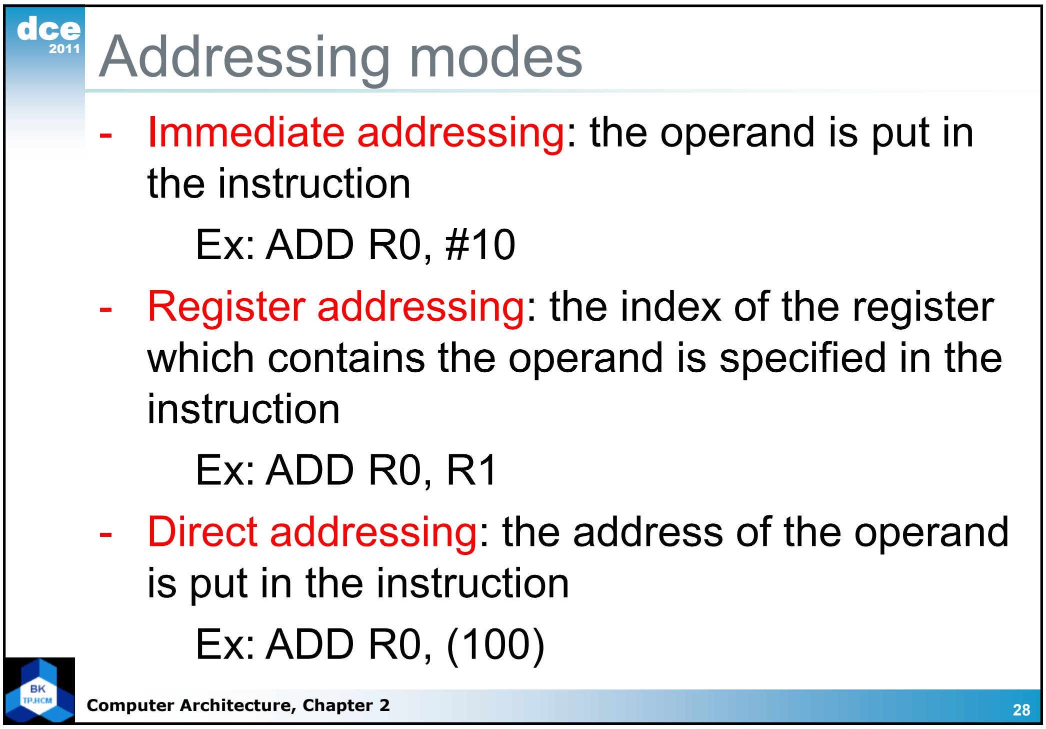


Addressing Modes

The way the processor refers to the operands is called addressing mode

The addressing modes can be classified based on:

- The source of data: Immediate, registers, memory
- The address calculation: Direct, indirect, indexed



Addressing modes

- Immediate addressing: the operand is put in the instruction

  Ex: ADD R0, #10

- Register addressing: the index of the register which contains the operand is specified in the instruction

  Ex: ADD R0, R1

- Direct addressing: the address of the operand is put in the instruction

  Ex: ADD R0, (100)

Addressing modes

- Register Indirect addressing: the address of the operand is put in the register which is specified in the instruction

  Ex: ADD R0, (R1)

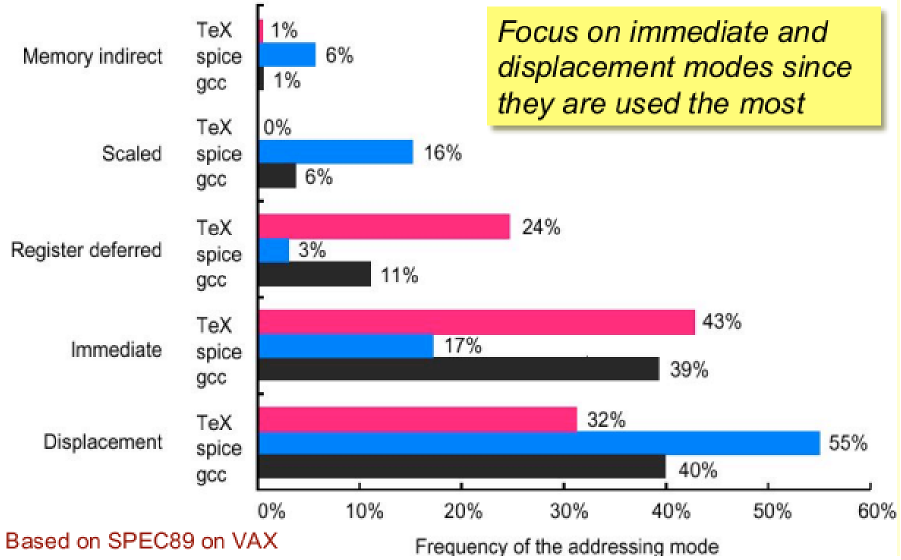
- Displacement addressing: the address of the operand is Base register + Displacement

  Ex: LD R1, 100(R2)

- Indexed addressing: The address of the operand is Base register + Indexed register

  Ex: ADD R3, (R1+R2)

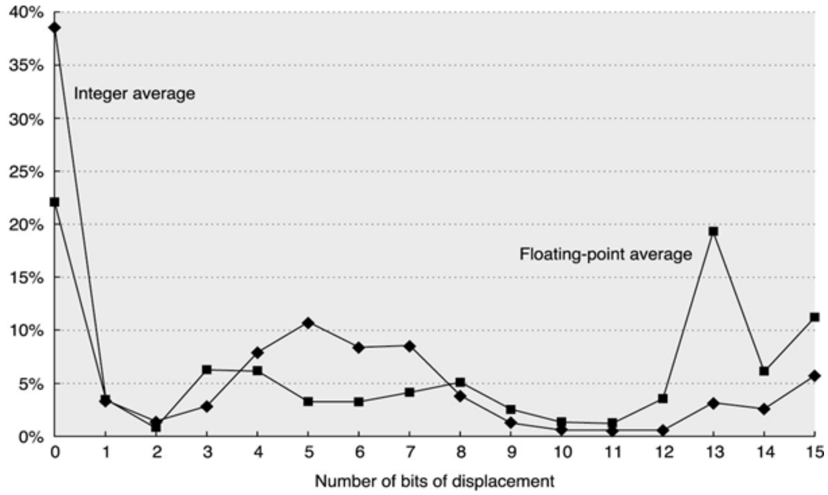
Addressing mode use



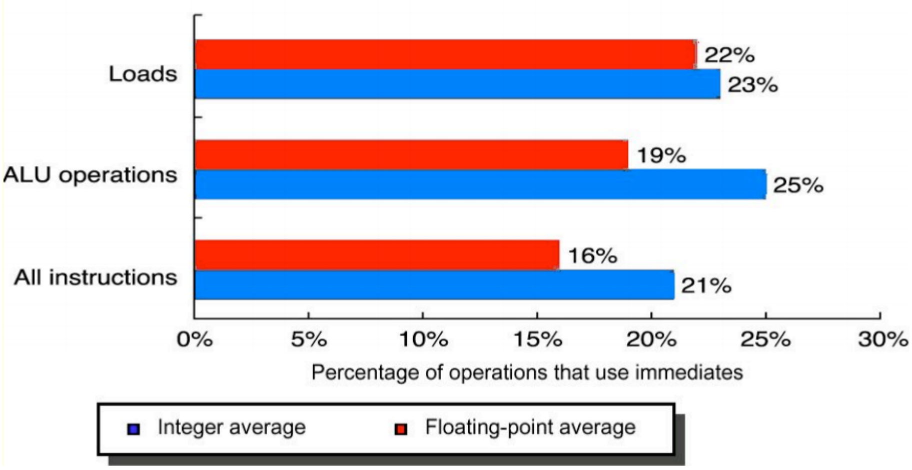
Focus on immediate and displacement modes since they are used the most

Based on SPEC89 on VAX

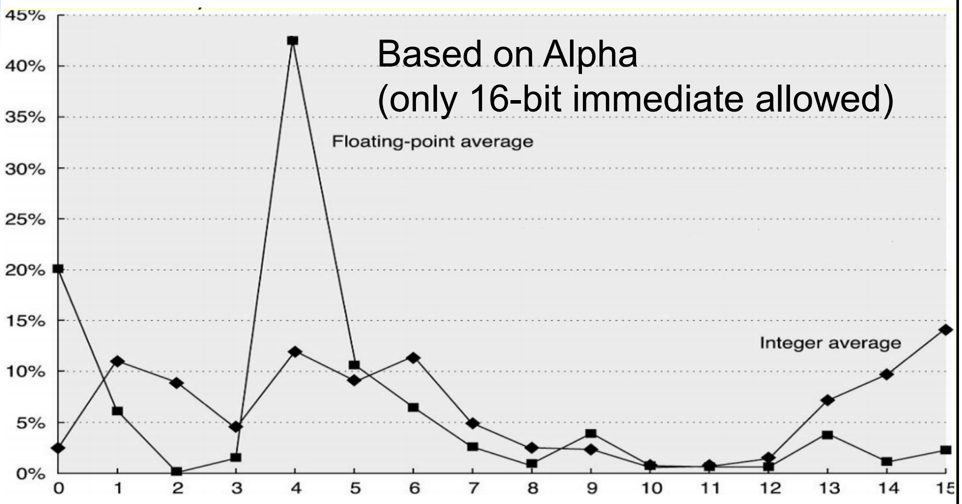
Addressing mode



Addressing mode



Addressing modes

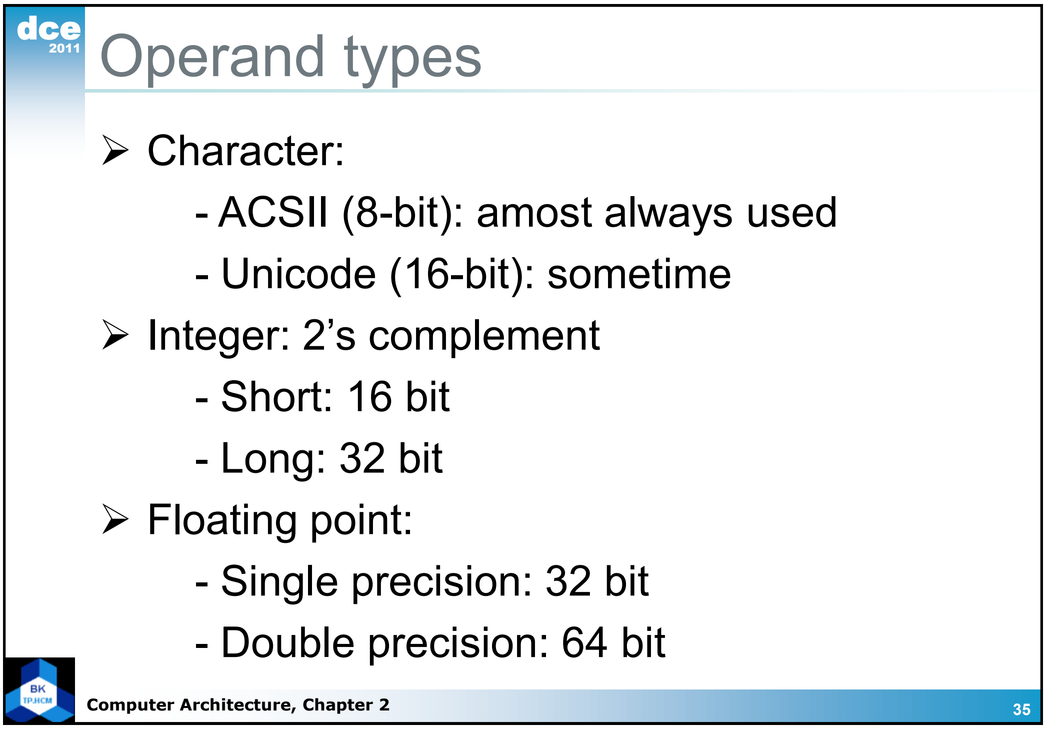


Based on Alpha
(only 16-bit immediate allowed)

- ➢ On VAX: 20%-25% longer than 16-bit

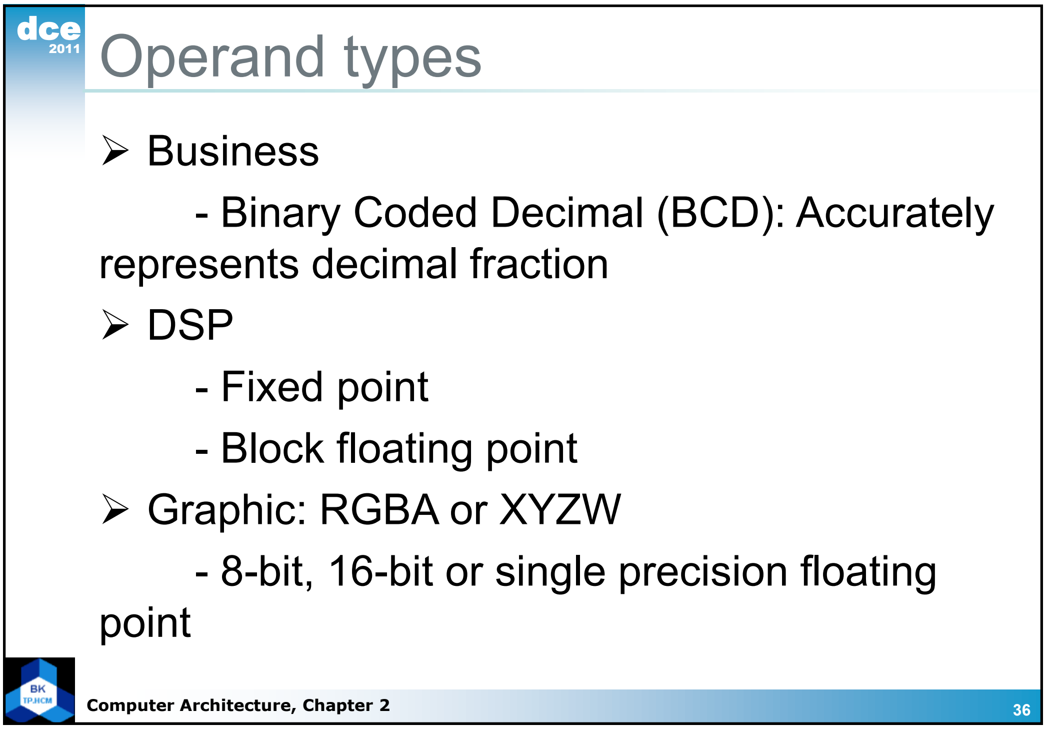
Addressing modes

- ➢ Is Memory indirect addressing necessary?
- ➢ Is Scaled addressing necessary?
- ➢ Is Register addressing necessary?
- ➢ How long should a displacement value be?
- ➢ How long should an immediate value be?



Operand types

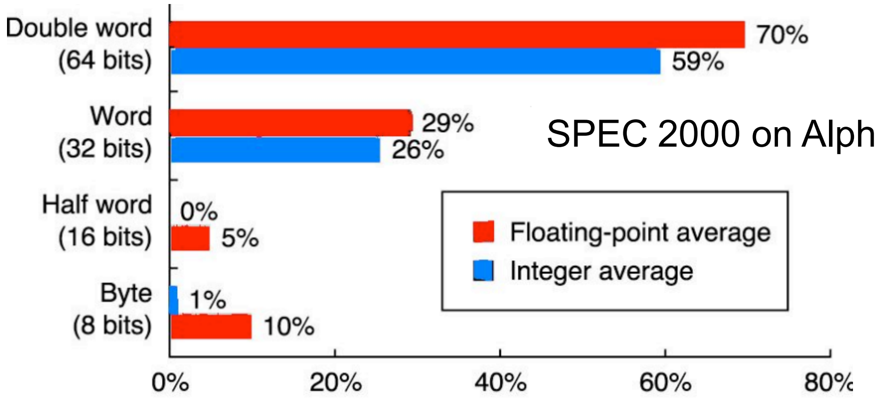
- Character:
  - ACSII (8-bit): amost always used
  - Unicode (16-bit): sometime
- Integer: 2's complement
  - Short: 16 bit
  - Long: 32 bit
- Floating point:
  - Single precision: 32 bit
  - Double precision: 64 bit



Operand types

- Business
  - Binary Coded Decimal (BCD): Accurately represents decimal fraction
- DSP
  - Fixed point
  - Block floating point
- Graphic: RGBA or XYZW
  - 8-bit, 16-bit or single precision floating point

Operand types & size

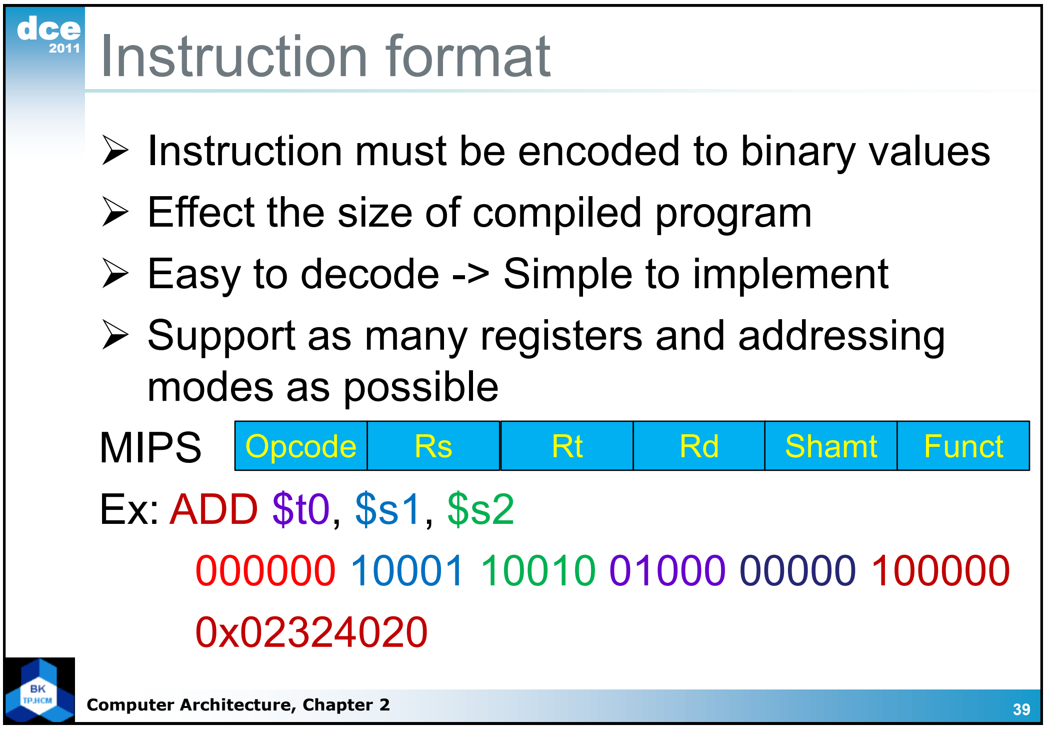


SPEC 2000 on Alpha

- Double word: double-precision floating point / address on 64-bit machine
- Word: integer / address in 32-bit machine

Operand types and size

- Should CPU support all those types of operand?
- Should CPU support very big-size operand?
- Is DSP’s data types used frequently?
- Is BCD used in most of operations?
- How about RGBA?



Instruction format

- Instruction must be encoded to binary values
- Effect the size of compiled program
- Easy to decode -> Simple to implement
- Support as many registers and addressing modes as possible

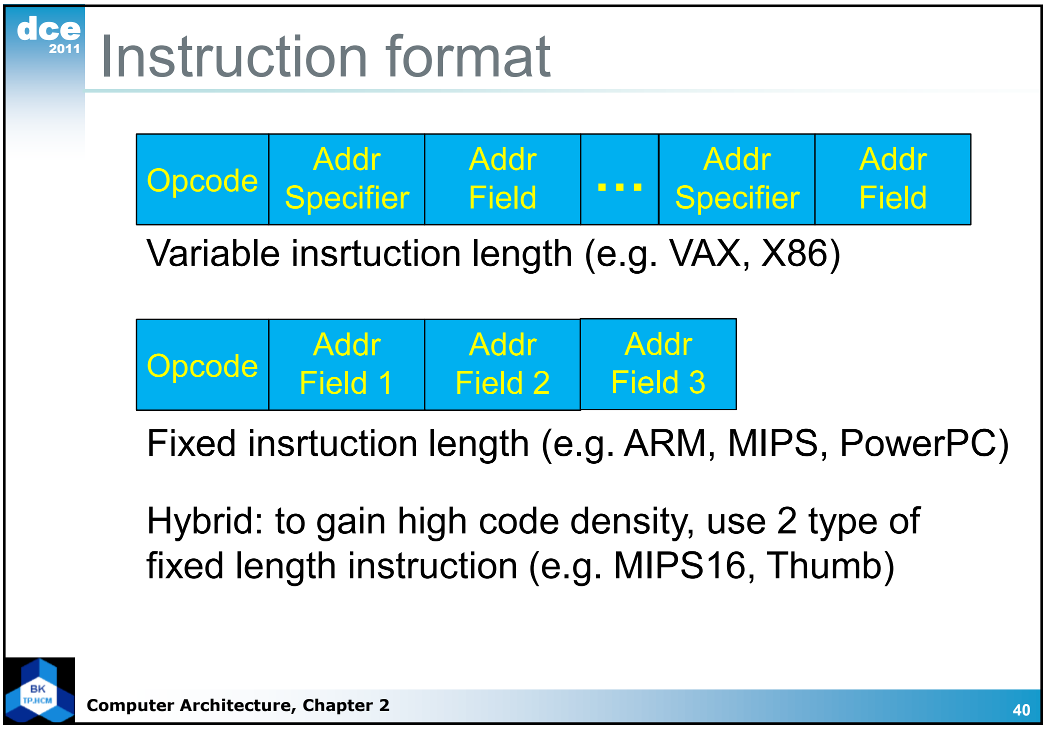
MIPS

Opcode	Rs	Rt	Rd	Shamt	Funct

Ex: ADD $t0, $s1, $s2

000000 10001 10010 01000 00000 100000

0x02324020



Instruction format

Opcode	Addr Specifier	Addr Field	...	Addr Specifier	Addr Field

Variable insrtuction length (e.g. VAX, X86)

Opcode	Addr Field 1	Addr Field 2	Addr Field 3

Fixed insrtuction length (e.g. ARM, MIPS, PowerPC)

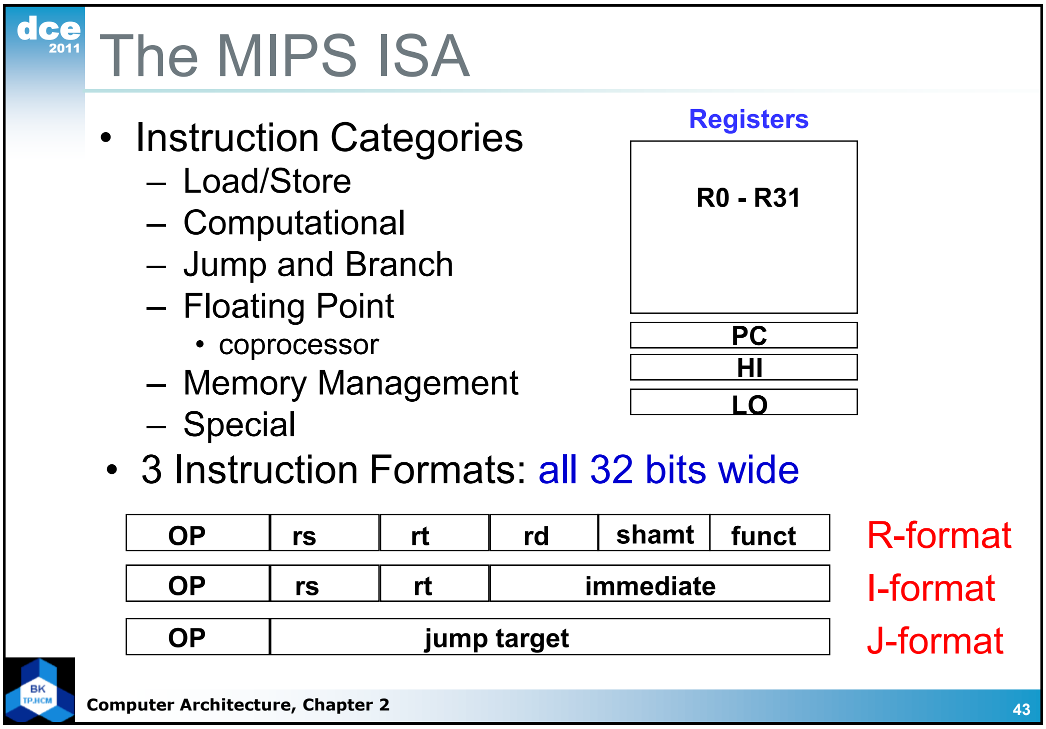
Hybrid: to gain high code density, use 2 type of fixed length instruction (e.g. MIPS16, Thumb)

Registers file

- ➢ Register is the fastest memory element
- ➢ Register cost much more than main memory
- ➢ Register is flexible for compiler to use
- ➢ More register need more bits to encode
- ➢ Register file with more locations can be slower
- ➢ How many locations in register file is the most effective?

Case study: MIPS

- Used as the example throughout the course
- Stanford MIPS commercialized by MIPS Technologies (www.mips.com)
- Large share of embedded core market
  - – Applications in consumer electronics, network/storage equipment, cameras, printers, …
- Typical of many modern ISAs



The MIPS ISA

- Instruction Categories
  - Load/Store
  - Computational
  - Jump and Branch
  - Floating Point
    - coprocessor
  - Memory Management
  - Special

Registers

R0 - R31

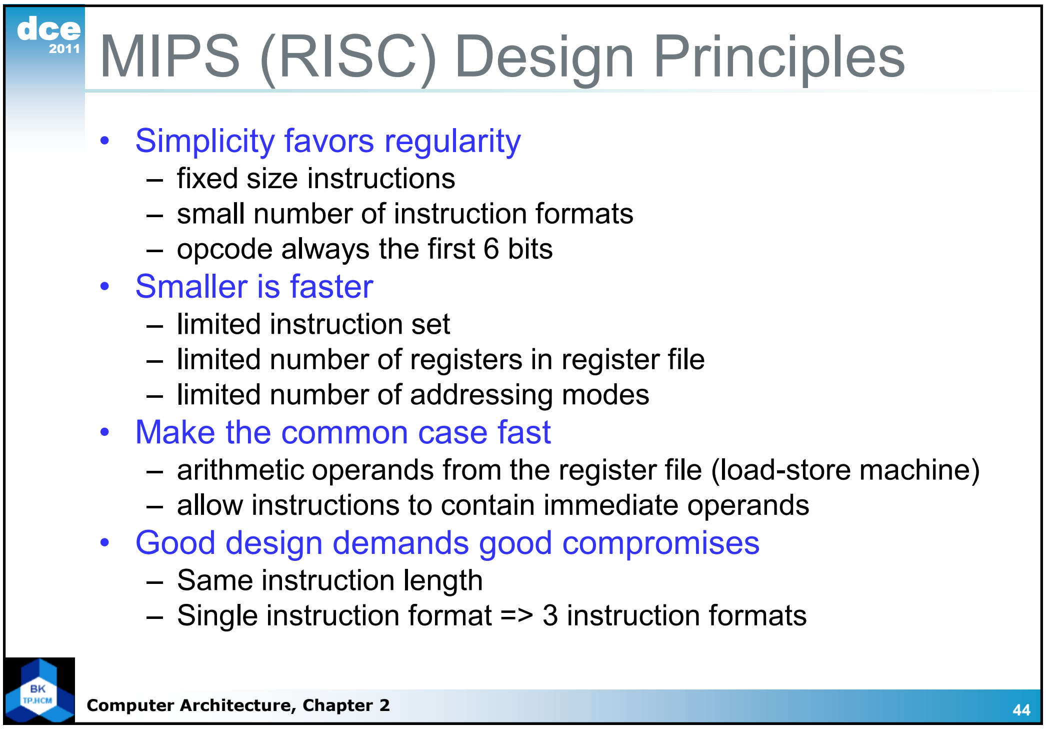
PC

HI

LO

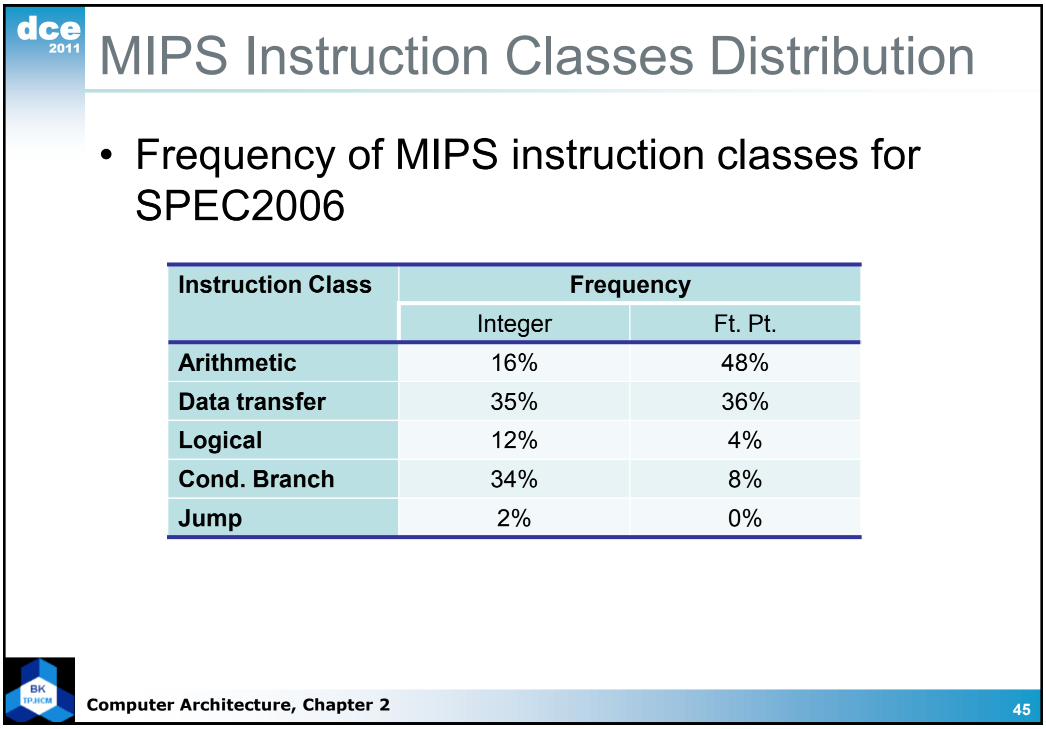
- 3 Instruction Formats: all 32 bits wide

OP	rs	rt	rd	shamt	funct	R-format
OP	rs	rt	immediate			I-format
OP	jump target					J-format



MIPS (RISC) Design Principles

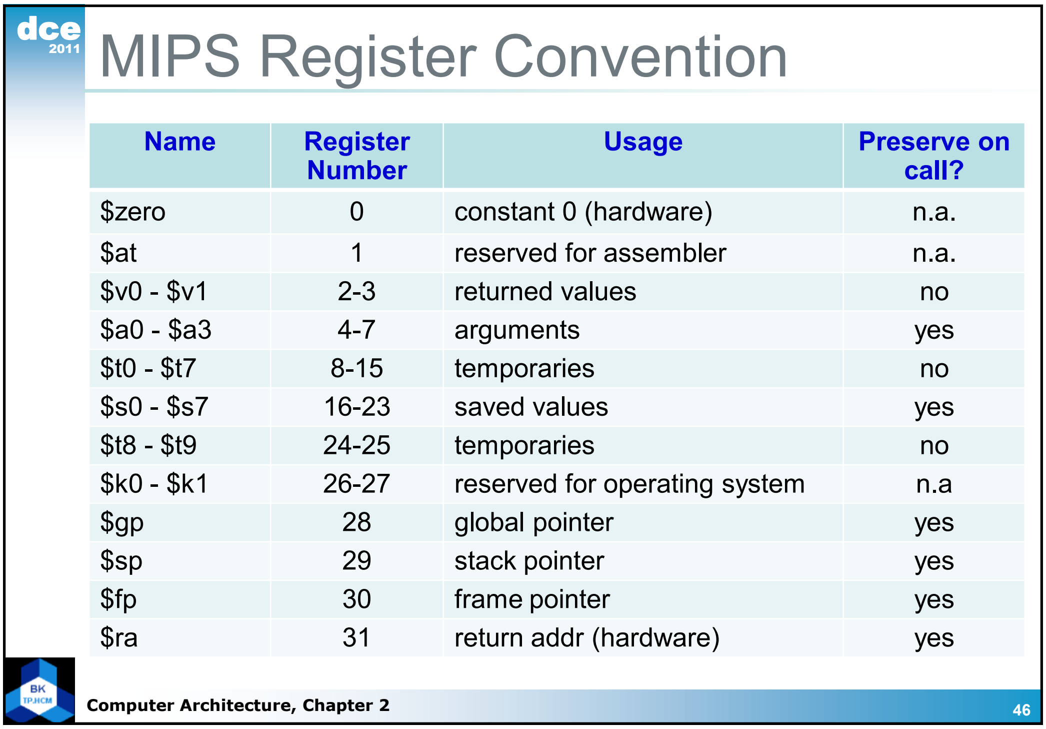
- Simplicity favors regularity
  - fixed size instructions
  - small number of instruction formats
  - opcode always the first 6 bits
- Smaller is faster
  - limited instruction set
  - limited number of registers in register file
  - limited number of addressing modes
- Make the common case fast
  - arithmetic operands from the register file (load-store machine)
  - allow instructions to contain immediate operands
- Good design demands good compromises
  - Same instruction length
  - Single instruction format => 3 instruction formats



MIPS Instruction Classes Distribution

- Frequency of MIPS instruction classes for SPEC2006

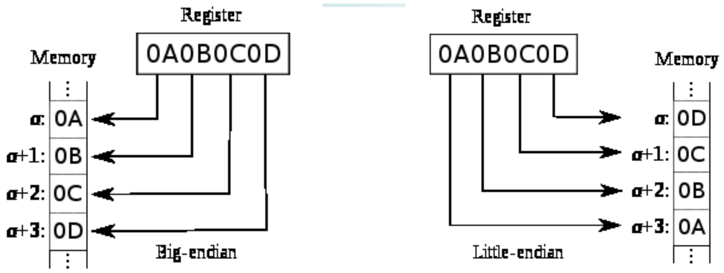
Instruction Class	Frequency	
	Integer	Ft. Pt.
Arithmetic	16%	48%
Data transfer	35%	36%
Logical	12%	4%
Cond. Branch	34%	8%
Jump	2%	0%



MIPS Register Convention

Name	Register Number	Usage	Preserve on call?
$zero	0	constant 0 (hardware)	n.a.
$at	1	reserved for assembler	n.a.
$v0 - $v1	2-3	returned values	no
$a0 - $a3	4-7	arguments	yes
$t0 - $t7	8-15	temporaries	no
$s0 - $s7	16-23	saved values	yes
$t8 - $t9	24-25	temporaries	no
$k0 - $k1	26-27	reserved for operating system	n.a
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return addr (hardware)	yes

MIPS - Endianness

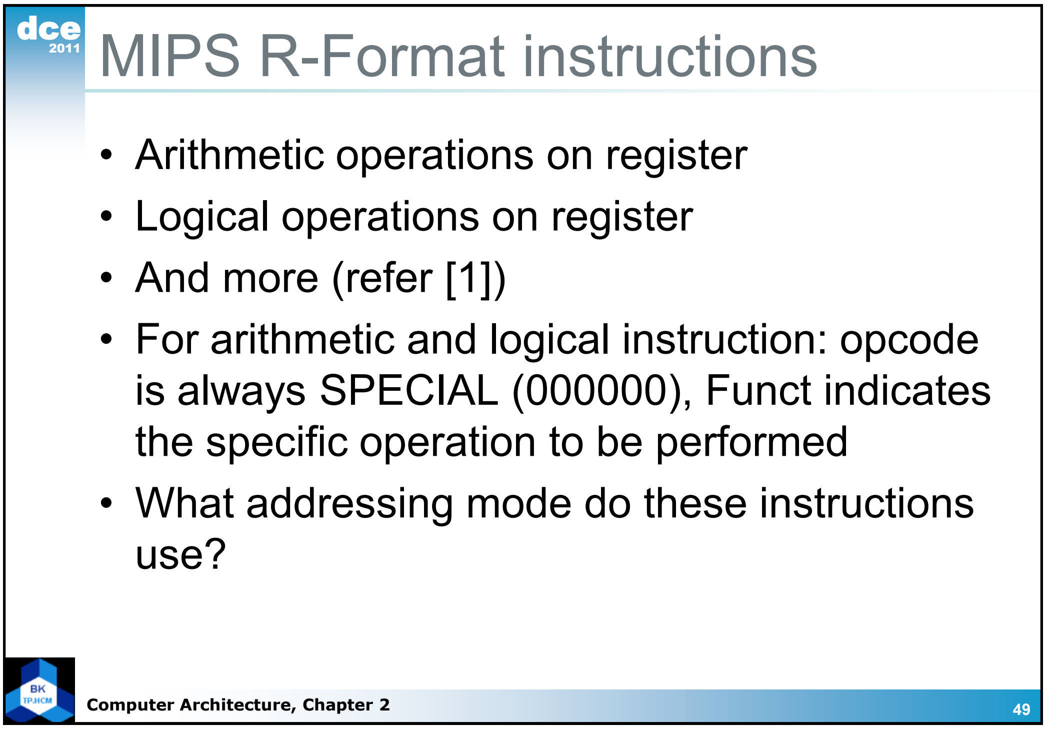


- Big Endian: Most-significant byte at lowest address of a word
- Little Endian: Least-significant byte at lowest address of a word
- MIPS is Big-endian

MIPS R-Format instructions

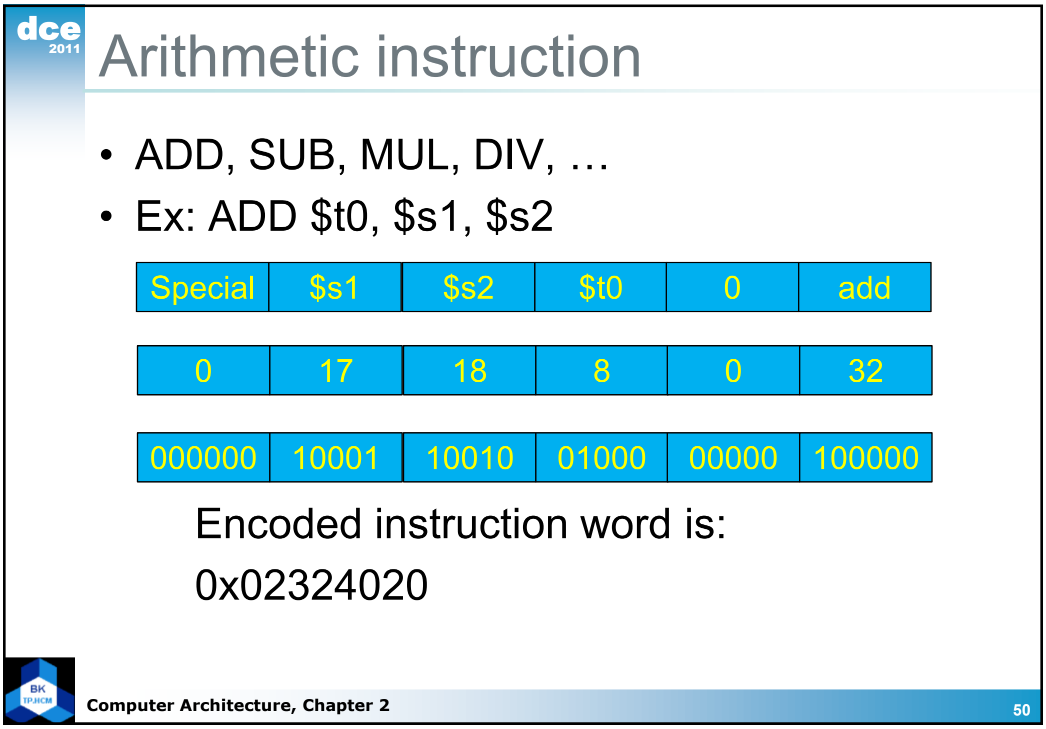
Op	Rs	Rt	Rd	Shamt	Funct

- Op: opcode
- Rs: First source register number
- Rt: Second source register number
- Rd: Destination register number
- Shamt: shift amount
  - Number of bit-shift –(left/right)
- Funct: Extend opcode
  - ALU function to encode the data path operation
    Execution: Rd <- Rs func Rt



MIPS R-Format instructions

- Arithmetic operations on register
- Logical operations on register
- And more (refer [1])
- For arithmetic and logical instruction: opcode is always SPECIAL (000000), Funct indicates the specific operation to be performed
- What addressing mode do these instructions use?



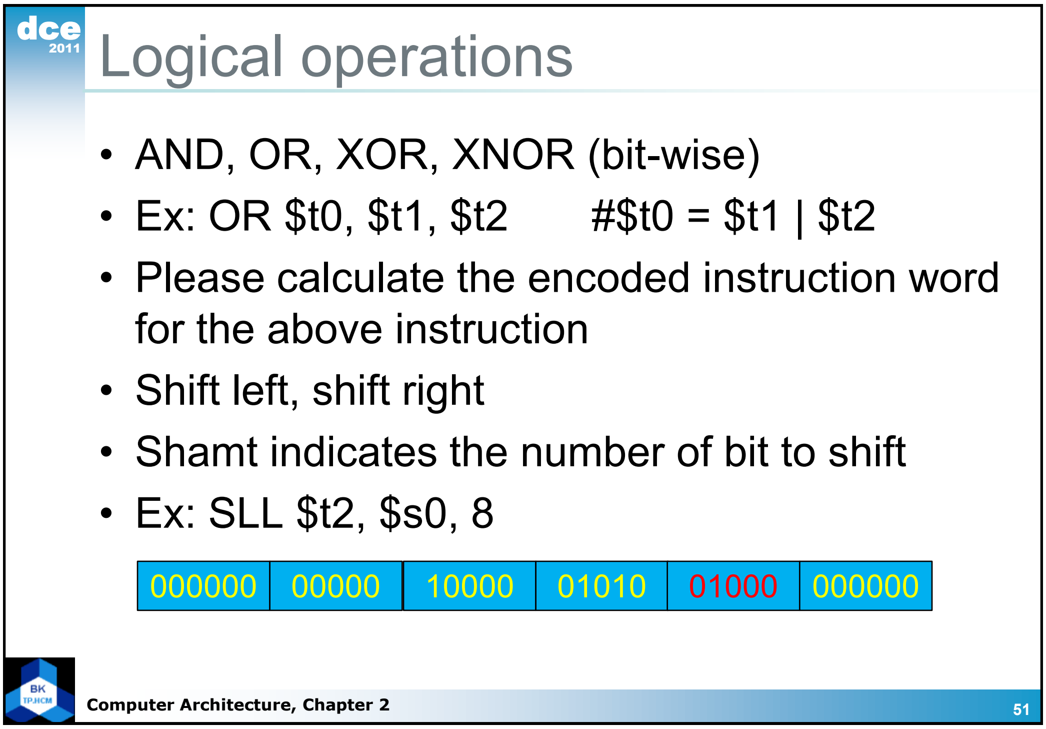
Arithmetic instruction

- ADD, SUB, MUL, DIV, …
- Ex: ADD $t0, $s1, $s2

Special	$s1	$s2	$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

Encoded instruction word is:

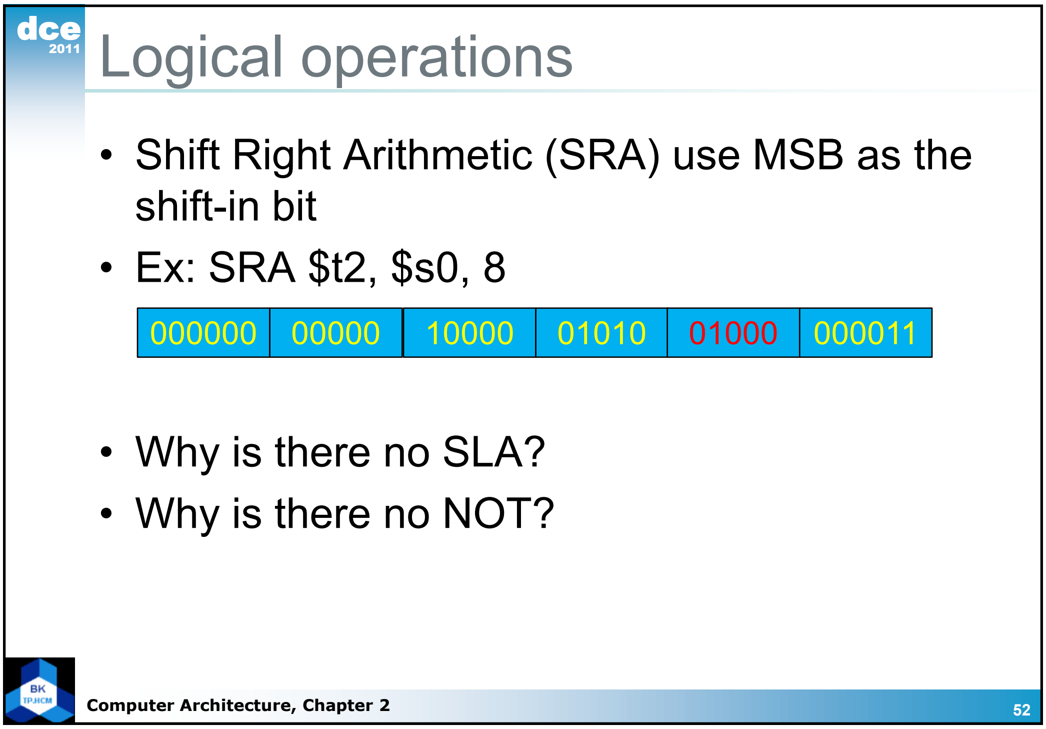
0x02324020



Logical operations

- AND, OR, XOR, XNOR (bit-wise)
- Ex: OR $t0, $t1, $t2   #$t0 = $t1 | $t2
- Please calculate the encoded instruction word for the above instruction
- Shift left, shift right
- Shamt indicates the number of bit to shift
- Ex: SLL $t2, $s0, 8

000000	00000	10000	01010	01000	000000



Logical operations

- Shift Right Arithmetic (SRA) use MSB as the shift-in bit
- Ex: SRA $t2, $s0, 8

000000	00000	10000	01010	01000	000011

- Why is there no SLA?
- Why is there no NOT?

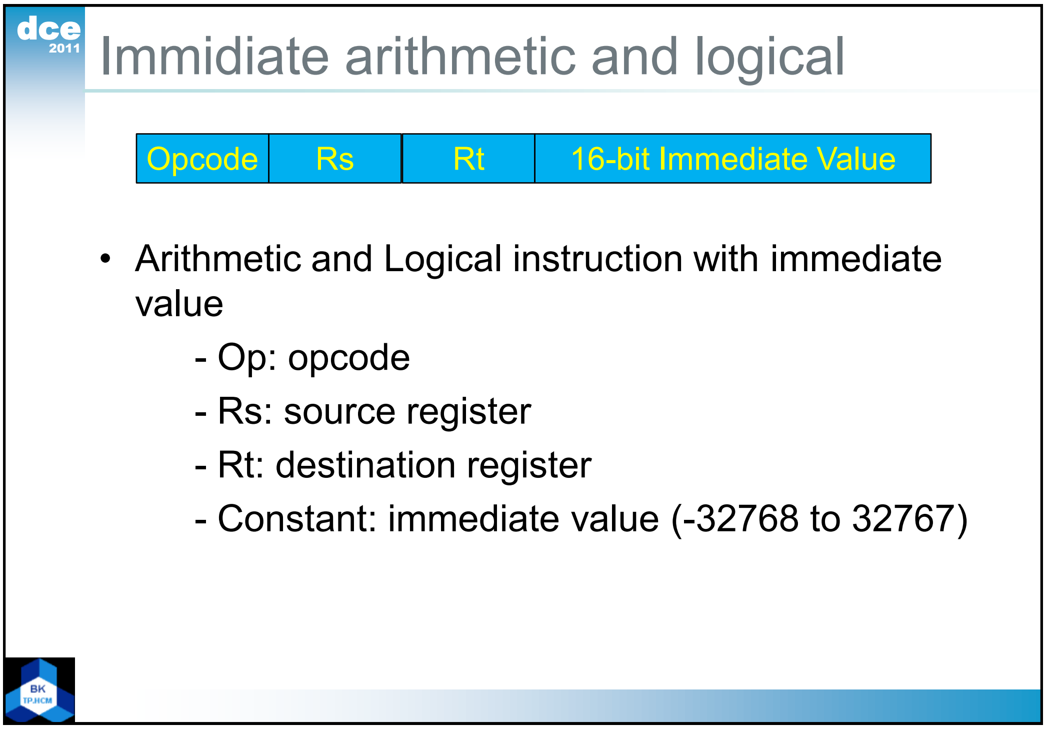
Jump register

- Register indirect addressing
- JR: Jump register
  - Rs: target address
  - Rd, Rt = 0; shamt: special purpose (hint) [1]
- JALR: Jump and link register
  - Rs: target address
  - Rd: return address
  - Rt = 0; shamt: special purpose (hint) [1]

MIPS I-Format instructions

Opcode	Rs	Rt	16-bit immediate value

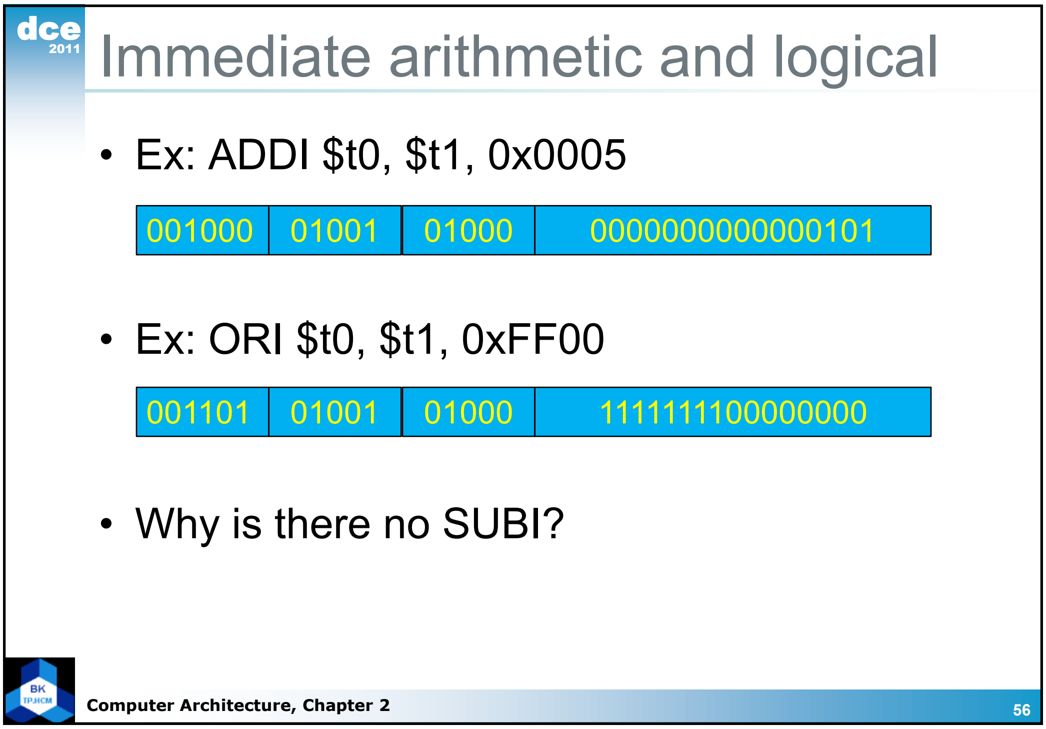
- This types of instruction can be:
  - Operation with immediate addressing
  - Operation with displacement addressing
  - Operation with PC-relative addressing



Immidiate arithmetic and logical

Opcode	Rs	Rt	16-bit Immediate Value

- Arithmetic and Logical instruction with immediate value
  - Op: opcode
  - Rs: source register
  - Rt: destination register
  - Constant: immediate value (-32768 to 32767)



Immediate arithmetic and logical

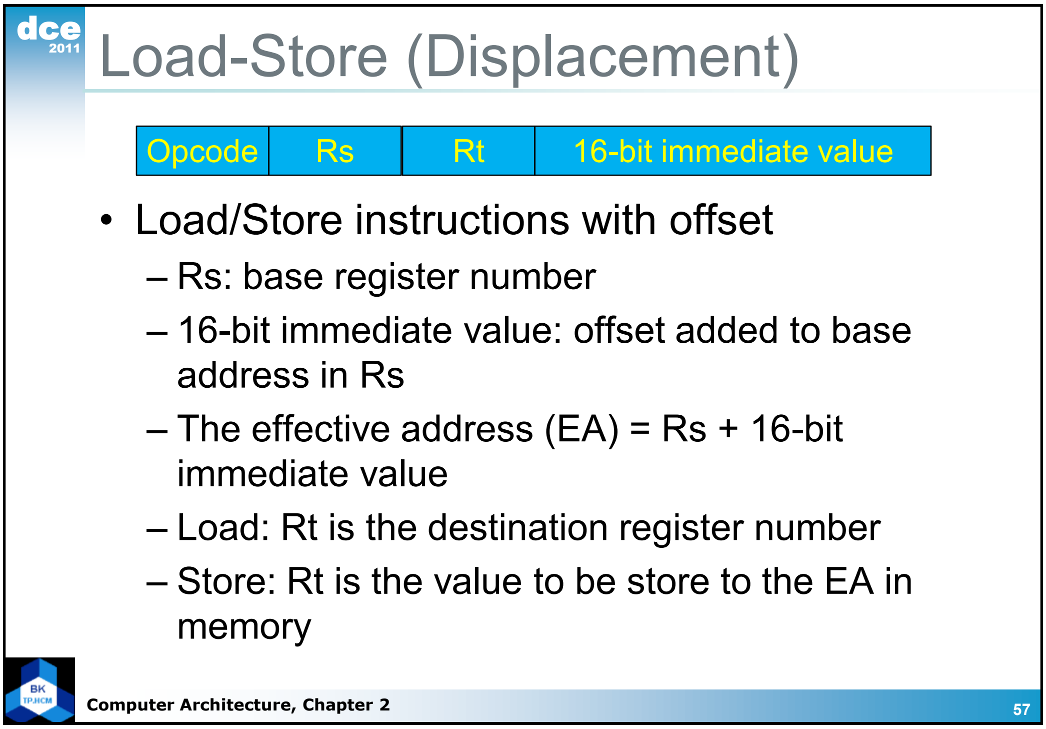
- Ex: ADDI $t0, $t1, 0x0005

001000	01001	01000	0000000000000101

- Ex: ORI $t0, $t1, 0xFF00

001101	01001	01000	1111111100000000

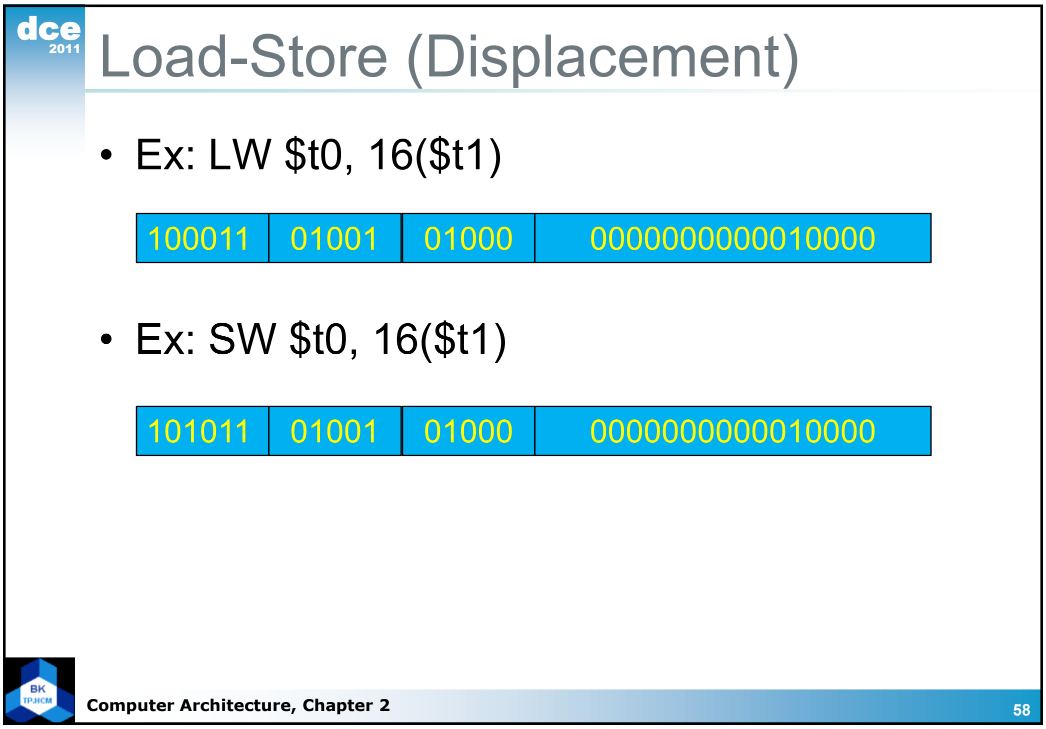
- Why is there no SUBI?



Load-Store (Displacement)

Opcode	Rs	Rt	16-bit immediate value

- Load/Store instructions with offset
  - Rs: base register number
  - 16-bit immediate value: offset added to base address in Rs
  - The effective address (EA) = Rs + 16-bit immediate value
  - Load: Rt is the destination register number
  - Store: Rt is the value to be store to the EA in memory



Load-Store (Displacement)

- Ex: LW $t0, 16($t1)

100011	01001	01000	0000000000010000

- Ex: SW $t0, 16($t1)

101011	01001	01000	0000000000010000

PC-Relative

Opcode	Rs	Rt	16-bit Immediate Value

- Near branch instructions
  - Rs: source register number
  - Rt: source register number
  - Target address = PC + offset x 4
  - PC already incremented by 4 by this time
- EX: BEQ $s0, $s1, 256
  - if($s0 == $s1) goto PC+256;

000100	10000	10001	0000000100000000

MIPS J-format Instructions

Opcode	26-bit Offset

- Jump (J and JAL)
- Pseudo-Direct addressing
  - Cannot put 32-bit value in instruction
  - Target address = $PC_{31\ldots28}$: (26-bit offset ×4)
- Ex: J 0x01000000

000010	00000001000000000000000000

Working with byte/halfword

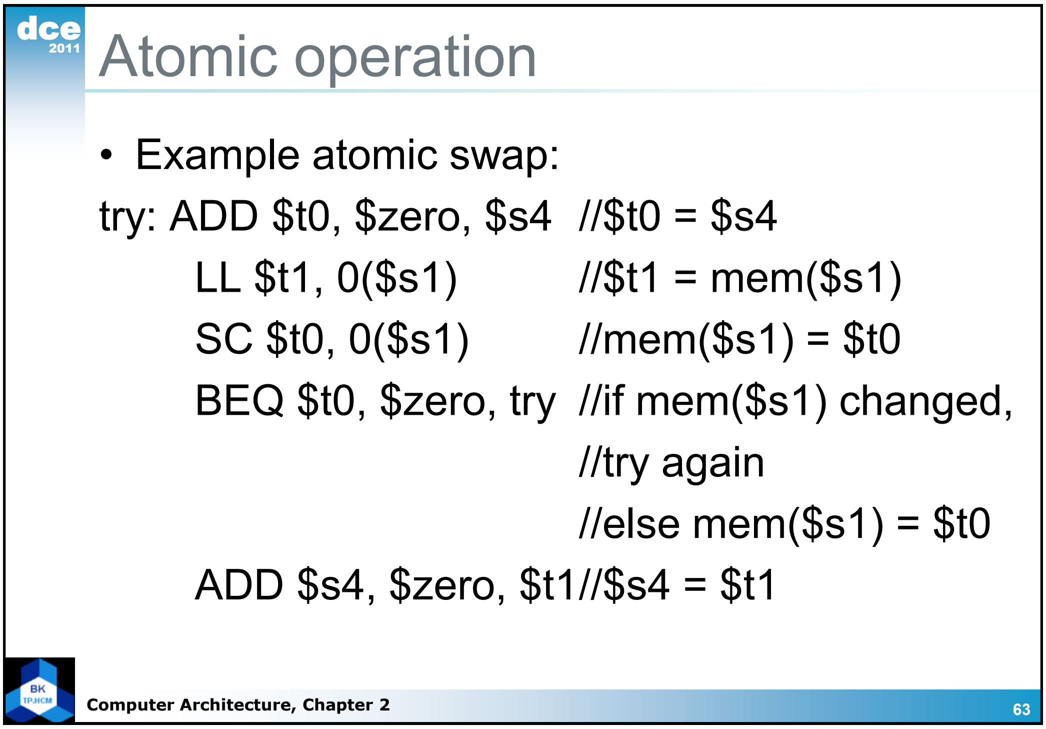
- LB/LH/LBU/LHU: Load byte/haftword from memory
  - – LBU $t0, 1($s0): Zero-extended
  - – LH $t0, 2($s0): Sign-extended
- SB, SH: Store byte/halfword to memory
  - – SB $t0, 1($s0)
  - – SH $t0, 2($s0)
- Why don't we have SBU, SHU?

Atomic operation

- An atomic Read-Modify-Write operation can be done by a pair of instructions: LL (Load Link Word) and SC (Store Conditional Word)

  LL $Rt, offset($Rs)

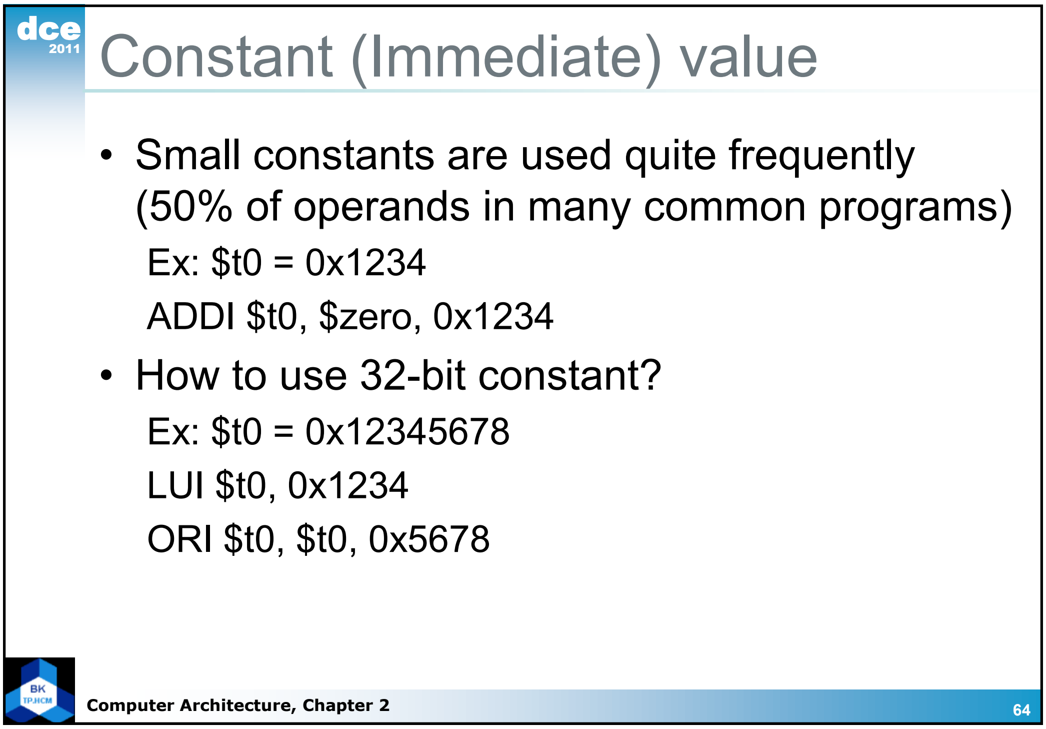
  SC $Rt, offset($Rs)
- If the content at memory address specified by LL is modified before SC to the same address, SC fails and return 0 in $Rt. Or else, SC store $Rt to memory and return 1 in $Rt



Atomic operation

- Example atomic swap:

```
try: ADD $t0, $zero, $s4  //$t0 = $s4
     LL $t1, 0($s1)       //$t1 = mem($s1)
     SC $t0, 0($s1)       //mem($s1) = $t0
     BEQ $t0, $zero, try  //if mem($s1) changed,
                          //try again
                          //else mem($s1) = $t0
     ADD $s4, $zero, $t1//$s4 = $t1
```



Constant (Immediate) value

- Small constants are used quite frequently (50% of operands in many common programs)

  Ex: $t0 = 0x1234

  ADDI $t0, $zero, 0x1234

- How to use 32-bit constant?

  Ex: $t0 = 0x12345678

  LUI $t0, 0x1234

  ORI $t0, $t0, 0x5678

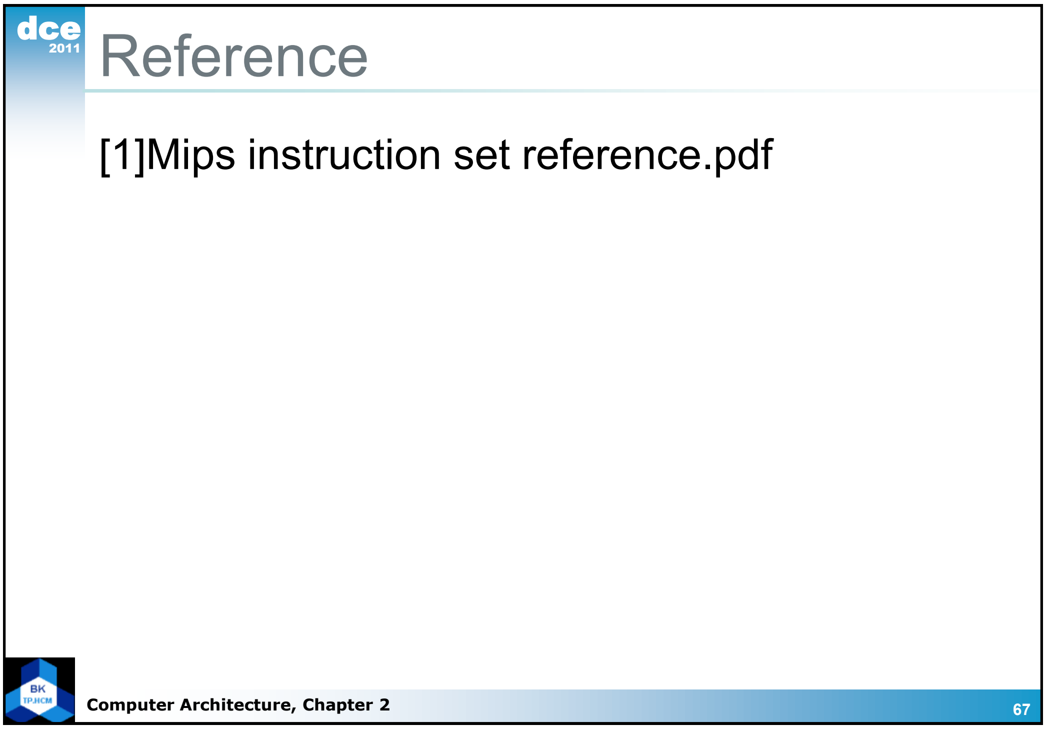
Procedure call

- Save return address
- Save necessary registers
- Callee execute the function
- Restore previously saved registers
- Restore return address
- Jump to the return address
  - JAL: Jump to a Label (Procedure), return address is stored in $ra (register 31)
  - JR: Jump to the address which is stored in a register

Procedure call: Factorial

- MIPS code:

```
fact:
    addi $sp, $sp, -8     # adjust stack for 2 items
    sw   $ra, 4($sp)      # save return address
    sw   $a0, 0($sp)      # save argument
    slti $t0, $a0, 1      # test for n < 1
    beq  $t0, $zero, L1
    addi $v0, $zero, 1    # if so, result is 1
    addi $sp, $sp, 8      #   pop 2 items from stack
    jr   $ra              #   and return
L1: addi $a0, $a0, -1     # else decrement n
    jal  fact             # recursive call
    lw   $a0, 0($sp)      # restore original n
    lw   $ra, 4($sp)      #   and return address
    addi $sp, $sp, 8      # pop 2 items from stack
    mul  $v0, $a0, $v0    # multiply to get result
    jr   $ra              # and return
```



Reference

[1]Mips instruction set reference.pdf