Advanced System Architectures

Assoc.Prof. Dr. Tran Ngoc Thinh
HCMC University of Technology

1

Review of Instructions Set Architecture

2

Outline

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

3

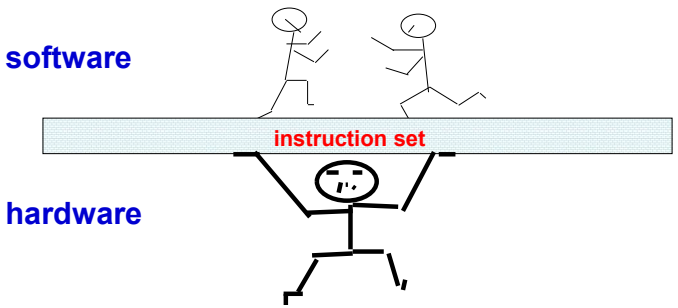
Machine Instruction

Computer can only understand binary values

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

4

The Instruction Set

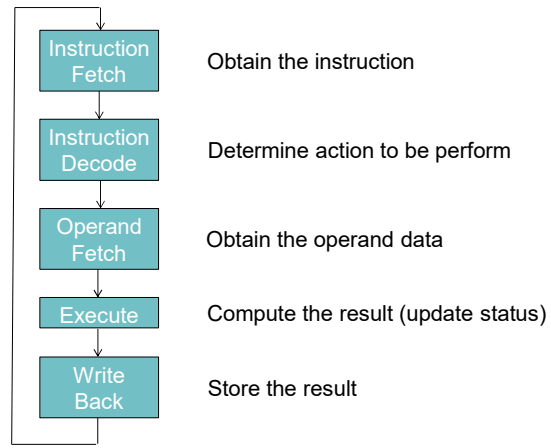


Instruction set: set of all instructions a processor can perform

Interface between software and hardware

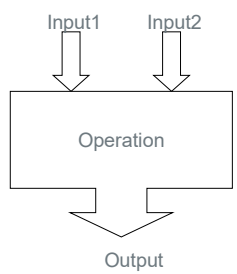
5

Instruction execution cycle



6

ISA Styles

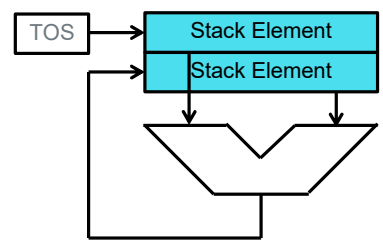


ISA Styles?

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

7

ISA Styles: Stack



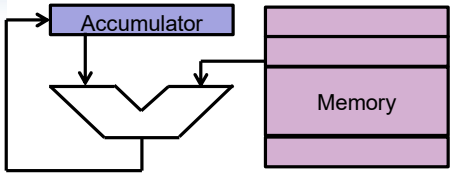
C= A+B?

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

8

ISA Styles: Accumulator



C= A+B?

LOAD A - *Put A in Accumulator (AC)*

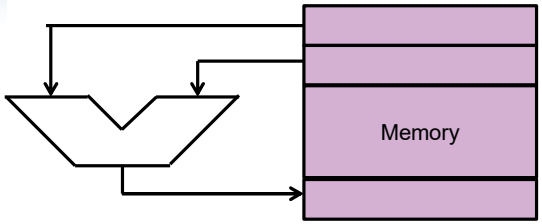
ADD B - *Add B with AC put result in AC*

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

9

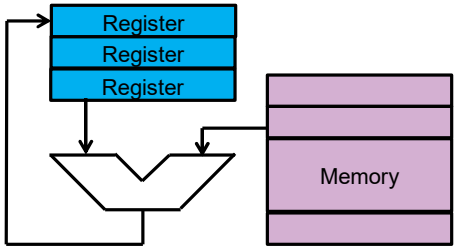
ISA Styles: Memory-memory



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

10

ISA Styles: Register-Memory



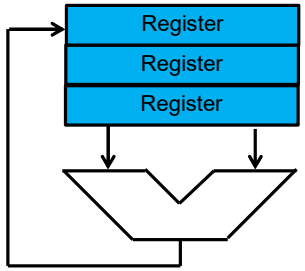
Input, Output: Register or Memory

C= A+B?

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

11

ISA Styles: Register-Register



C= A+B?

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

- Register-Register: All operations are on registers
- Need specific Load and Store instruction to access memory

12

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

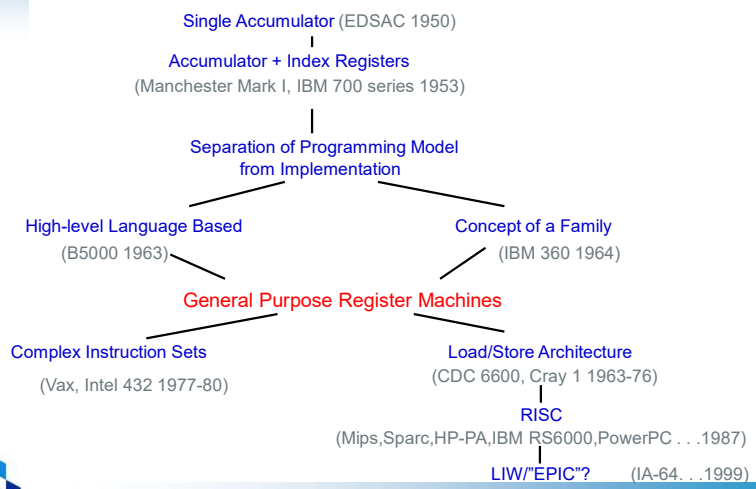
13

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

14

Evolution of Instruction Sets



15

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 **Thanh ghi là dạng bộ nhớ tốc độ cao**

16

Simple format

Opcode	Source Operand reference	Result 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

17

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

18

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

19

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

20

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

Addressing Modes

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

The addressing modes can be classified based on:

gt tức thời (= constant)

- 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

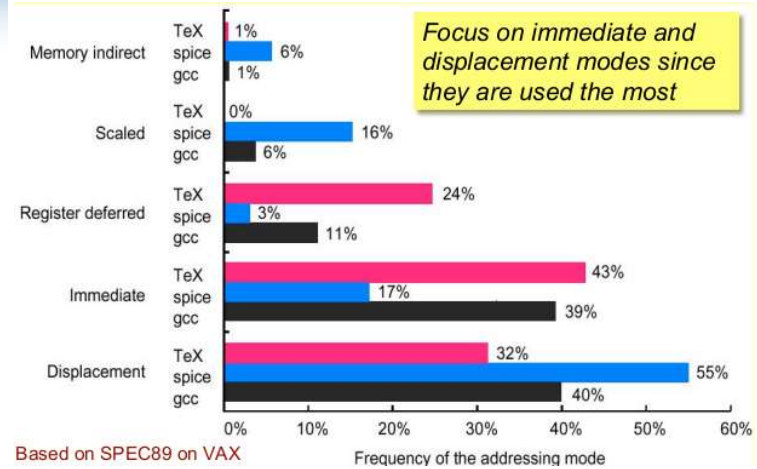
  Ex: LD R1, 100(R2)

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

  Ex: ADD R3, (R1+R2)

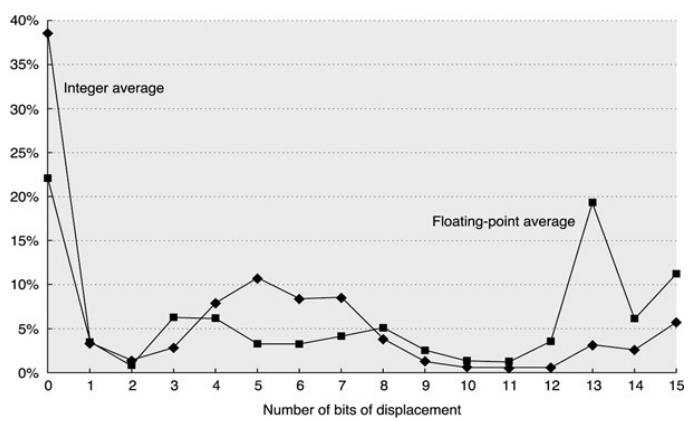
25

Addressing mode use



26

Addressing mode



27

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?

28

Addressing modes

Immediate	add r1 = r2 + 5
Register	add r1 = r2 + r3
Direct	load r1 = M [4000]
Register Indirect	add r1 = r2 + M[r2]
Displacement	load r1 = M[r2 + 4000]
Indexed/Base	add r1 = r3 + M[r2 + r3]
Memory Indirect	load r1 = M[M[r2]]
PC Relative	branch r1 < r3, 1000
Scaled	load r1 = M[100 + r3 + r4 * d]

29

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

30

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

31

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?

32

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 instruction length (e.g. VAX, X86)

Opcode	Addr Field 1	Addr Field 2	Addr Field 3

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

Hybrid: to gain high code density, use 2 types 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?

Complex Instruction Set Computer (CISC)

- Emphasizes doing more with each instruction:
  - Thus fewer instructions per program (more compact code).
- Motivated by the high cost of memory and hard disk capacity when original CISC architectures were proposed
  - When M6800 was introduced: 16K RAM = $500, 40M hard disk = $ 55, 000
  - When MC68000 was introduced: 64K RAM = $200, 10M HD = $5,000
- Original CISC architectures evolved with faster more complex CPU designs but backward instruction set compatibility had to be maintained.
- Wide variety of addressing modes:
  - 14 in MC68000, 25 in MC68020
- A number instruction modes for the location and number of operands:
  - The VAX has 0- through 3-address instructions.
- Variable-length instruction encoding.

Example CISC ISA: Motorola 680X0

18 addressing modes:

- Data register direct.
- Address register direct.
- Immediate.
- Absolute short.
- Absolute long.
- Address register indirect.
- Address register indirect with postincrement.
- Address register indirect with predecrement.
- Address register indirect with displacement.
- Address register indirect with index (8-bit).
- Address register indirect with index (base).
- Memory inderect postindexed.
- Memory indirect preindexed.
- Program counter indirect with index (8-bit).
- Program counter indirect with index (base).
- Program counter indirect with displacement.
- Program counter memory indirect postindexed.
- Program counter memory indirect preindexed.

GPR ISA (Register-Memory)

Operand size:

- **Range from 1 to 32 bits, 1, 2, 4, 8, 10, or 16 bytes.**

Instruction Encoding:

- **Instructions are stored in 16-bit words.**
- **the smallest instruction is 2-bytes (one word).**
- **The longest instruction is 5 words (10 bytes) in length.**

37

Example CISC ISA:Intel IA-32, X86 (80386)

12 addressing modes:

- Register.
- Immediate.
- Direct.
- Base.
- Base + Displacement.
- Index + Displacement.
- Scaled Index + Displacement.
- Based Index.
- Based Scaled Index.
- Based Index + Displacement.
- Based Scaled Index + Displacement.
- Relative.

GPR ISA (Register-Memory)

Operand sizes:

- **Can be 8, 16, 32, 48, 64, or 80 bits long.**
- **Also supports string operations.**

Instruction Encoding:

- **The smallest instruction is one byte.**
- **The longest instruction is 12 bytes long.**
- **The first bytes generally contain the opcode, mode specifiers, and register fields.**
- **The remainder bytes are for address displacement and immediate data.**

38

Reduced Instruction Set Computer (RISC)

- Focuses on reducing the number and complexity of instructions of the machine.
- Reduced CPI. Goal: At least one instruction per clock cycle.
- Designed with pipelining in mind.
- Fixed-length instruction encoding. **(CPI = 1 or less)** **Siêu pipeline hoặc super scala có CPI < 1**
- Only load and store instructions access memory.
- Simplified addressing modes. **(Thus more instructions executed than CISC)**
  - Usually limited to immediate, register indirect, register displacement, indexed.
- Delayed loads and branches.
- Instruction pre-fetch and speculative execution.
- Examples: MIPS, SPARC, PowerPC, Alpha

39

Example RISC ISA: HP Precision Architecture, HP PA-RISC

7 addressing modes:

- Register
- Immediate
- Base with displacement
- Base with scaled index and displacement
- Predecrement
- Postincrement
- PC-relative

Operand sizes:

- **Five operand sizes ranging in powers of two from 1 to 16 bytes.**

Instruction Encoding:

- **Instruction set has 12 different formats.**
- **All are 32 bits in length.**

40

Example RISC ISA: Alpha AXP

4 addressing modes:

- Register direct.
- Immediate.
- Register indirect with displacement.
- PC-relative.

Operand sizes:

- **Four operand sizes: 1, 2, 4 or 8 bytes.**

Instruction Encoding:

- **Instruction set has 7 different formats.**
- **All are 32 bits in length.**

41

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
- GroupX_RISC_CISC_assignment_K2021.doc
- Each Group has 4-6 persons

42

The MIPS ISA

- Instruction Categories
  - Load/Store
  - Computational
  - Jump and Branch
  - Floating Point
    - coprocessor
  - Memory Management
  - Special
- 3 Instruction Formats: all 32 bits wide

Registers

R0 - R31
PC
HI
LO

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

43

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

44

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%

45

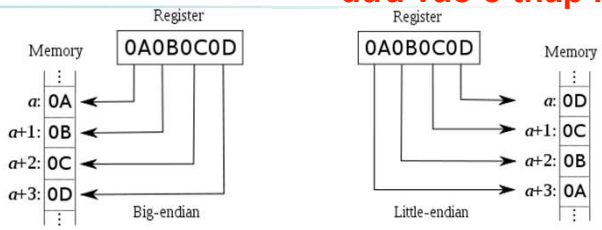
MIPS Register Convention

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

46

MIPS - Endianness

0A: byte cao nhất, đưa vào ô thấp là a



- Big Endian: Most-significant byte at lowest address of a word **Byte cao ở vị trí thấp**
- Little Endian: Least-significant byte at lowest address of a word
- MIPS is Big-endian

47

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

48

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? **Lệnh NOT = A NOR 0**

Jump register

- Register indirect addressing
- JR: Jump register (JR rs)
  - Rs: target address
  - Rd, Rt = 0; shamt: special purpose (hint) [1]
- JALR: Jump and link register (JALR rs, JALR rd,rs)
  - 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? **ADDI với số âm là được**

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

57

Load-Store (Displacement)

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

100011	01001	01000	0000000000010000

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

101011	01001	01000	0000000000010000

58

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+256x4;

000100	10000	10001	0000000100000000

59

MIPS J-format Instructions

Opcode	26-bit Offset

- Jump (J and JAL)
- Pseudo-Direct addressing
  - Cannot put 32-bit value in instruction
  - Target address = $PC_{31...28}$: (26-bit offset ×4)
- Ex: J 0x01000000

000010	00000001000000000000000000000000

60

Quiz

- How to move data from $t0 to $t1 using 1 MIPS instruction? **Cộng t0 với thanh ghi 0**

61

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) **Store byte/halfword không**
  - SH $t0, 2($s0) **quan tâm tới dấu**
- Why don't we have SBU, SHU?

62

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

63

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
```


64

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

65

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

66

Example in C: swap

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

- **Assume swap is called as a procedure**
- **Assume temp is register $15; arguments in $a1, $a2; $16 is scratch reg:**
- **Write MIPS code**

67

swap: MIPS

```
swap:
   addi $sp,$sp, –4      ; create space on stack
   sw   $16, 0($sp)      ; callee saved register put onto stack
   sll  $t2, $a2,2       ; multiply k by 4
   addu $t2, $a1,$t2     ; address of v[k]
   lw   $15, 0($t2)      ; load v[k]
   lw   $16, 4($t2)      ; load v[k+1]
   sw   $16, 0($t2)      ; store v[k+1] into v[k]
   sw   $15, 4($t2)      ; store old value of v[k] into v[k+1]
   lw   $16, 0($sp)      ; callee saved register restored from stack
   addi $sp,$sp, 4       ; restore top of stack
   jr   $31              ; return to place that called swap
```

68

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 n*fact(n-1)
    jr   $ra               # and return
```


69

Quiz

- Which one, caller or callee, has to save return address? **chương trình con (callee). Ctr cha vẫn đc**
- Which one, caller or callee, has to save necessary registers? **con. Nhưng con or cha đều được**
- Is it necessary for leaf-procedure to save return address (in MIPS)?

Không cần lưu

70

MIPS arithmetic instructions

Instruction	*Example*	*Meaning*	*Comments*
add	add $1,$2,$3	$1 = $2 + $3	3 operands; exception possible
subtract	sub $1,$2,$3	$1 = $2 – $3	3 operands; exception possible
add immediate	addi $1,$2,100	$1 = $2 + 100	+ constant; exception possible
add unsigned	addu $1,$2,$3	$1 = $2 + $3	3 operands; no exceptions
subtract unsigned	subu $1,$2,$3	$1 = $2 – $3	3 operands; no exceptions
add imm. unsign.	addiu $1,$2,100	$1 = $2 + 100	+ constant; no exceptions
multiply	mult $2,$3	Hi, Lo = $2 x $3	64-bit signed product
multiply unsigned	multu$2,$3	Hi, Lo = $2 x $3	64-bit unsigned product
divide	div $2,$3	Lo = $2 ÷ $3, Hi = $2 mod $3	Lo = quotient, Hi = remainder
divide unsigned	divu $2,$3	Lo = $2 ÷ $3, Hi = $2 mod $3	Unsigned quotient & remainder
Move from Hi	mfhi $1	$1 = Hi	Used to get copy of Hi
Move from Lo	mflo $1	$1 = Lo	Used to get copy of Lo

71

MIPS logical instructions

Instruction	*Example*	*Meaning*	*Comment*
and	and $1,$2,$3	$1 = $2 & $3	3 reg. operands; Logical AND
or	or $1,$2,$3	$1 = $2 | $3	3 reg. operands; Logical OR
xor	xor $1,$2,$3	$1 = $2 ⊕ $3	3 reg. operands; Logical XOR
nor	nor $1,$2,$3	$1 = ~($2 |$3)	3 reg. operands; Logical NOR
and immediate	andi $1,$2,10	$1 = $2 & 10	Logical AND reg, constant
or immediate	ori $1,$2,10	$1 = $2 | 10	Logical OR reg, constant
xor immediate	xori $1, $2,10	$1 = ~$2 &~10	Logical XOR reg, constant
shift left logical	sll $1,$2,10	$1 = $2 << 10	Shift left by constant
shift right logical	srl $1,$2,10	$1 = $2 >> 10	Shift right by constant
shift right arithm.	sra $1,$2,10	$1 = $2 >> 10	Shift right (sign extend)
shift left logical	sllv $1,$2,$3	$1 = $2 << $3	Shift left by variable
shift right logical	srlv $1,$2, $3	$1 = $2 >> $3	Shift right by variable
shift right arithm.	srav $1,$2, $3	$1 = $2 >> $3	Shift right arith. by variable

72

MIPS data transfer instructions

Instruction	*Comment*
SW 500(R4), R3	Store word
SH 502(R2), R3	Store half
SB 41(R3), R2	Store byte
LW R1, 30(R2)	Load word
LH R1, 40(R3)	Load halfword
LHU R1, 40(R3)	Load halfword unsigned
LB R1, 40(R3)	Load byte
LBU R1, 40(R3)	Load byte unsigned
LUI R1, 40	Load Upper Immediate (16 bits shifted left by 16)

73

Compare and Branch

- Compare and Branch PC- BEQ rs, rt, offset if R[rs] == R[rt] then relative branch
  BNE rs, rt, offset <>
- BLEZ rs, offset if R[rs] <= 0 then PC-relative branch
- BGTZ rs, offset >
- BLTZ rs, offset <
- BGEZ rs, offset >=

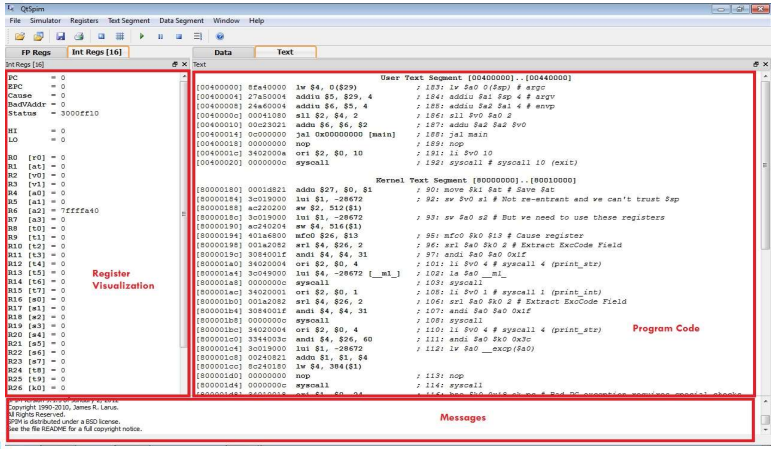
74

MIPS jump, branch, compare instructions

Instruction	*Example*	*Meaning*
branch on equal	beq $1,$2,100	if ($1 == $2) go to PC+4+100
	Equal test; PC relative branch	
branch on not eq.	bne $1,$2,100	if ($1!= $2) go to PC+4+100
	Not equal test; PC relative	
set on less than	slt $1,$2,$3	if ($2 < $3) $1=1; else $1=0
	Compare less than; 2's comp.	
set less than imm.	slti $1,$2,100	if ($2 < 100) $1=1; else $1=0
	Compare < constant; 2's comp.	
set less than uns.	sltu $1,$2,$3	if ($2 < $3) $1=1; else $1=0
	Compare less than; natural no.	
set l. t. imm. uns.	sltiu $1,$2,100	if ($2 < 100) $1=1; else $1=0
	Compare < constant; natural	
jump	j 10000	go to 10000
	Jump to target address	
jump register	jr $31	go to $31
	For switch, procedure return	
jump and link	jal 10000	$31 = PC + 4; go to 10000
	For procedure call	

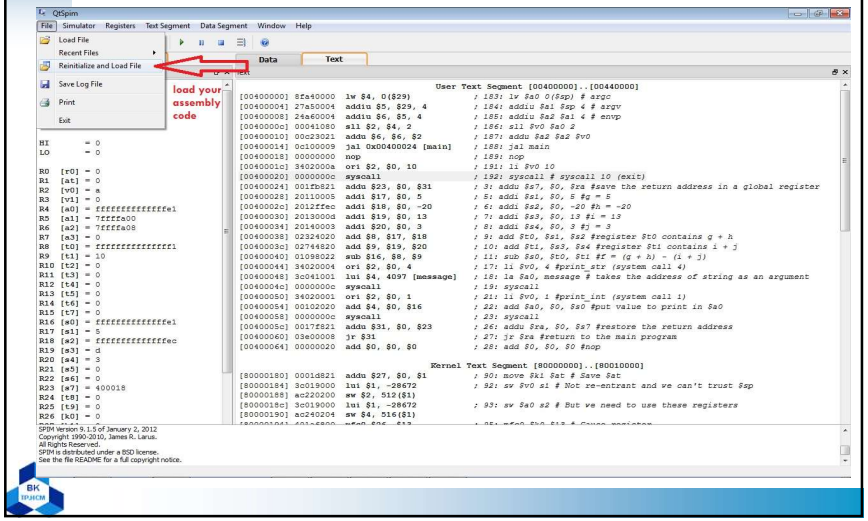
75

Start SPIM



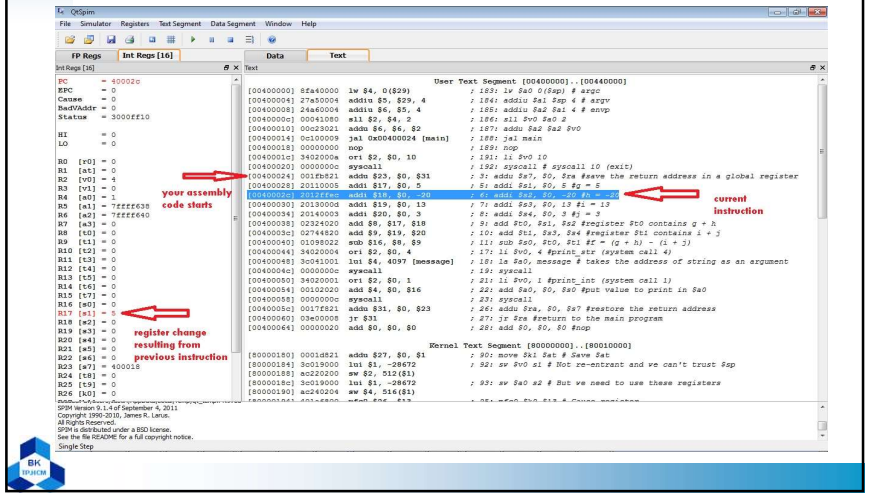
76

Load Program



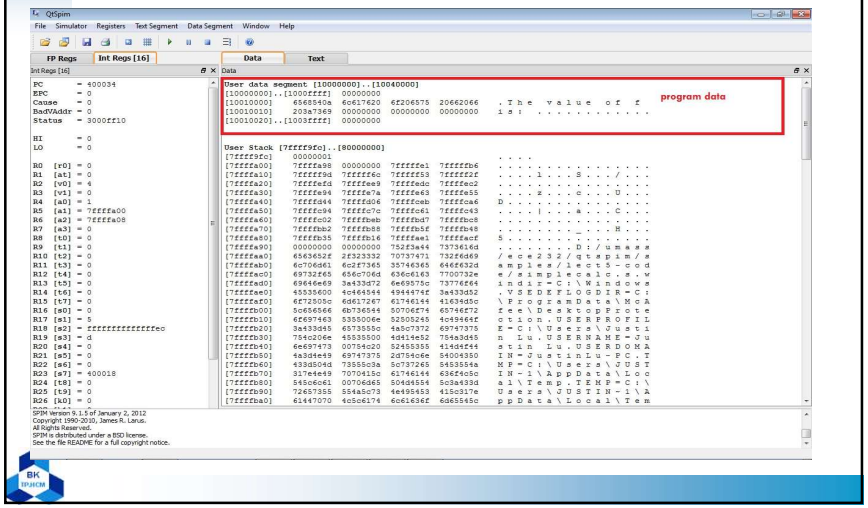
77

Execute Program



78

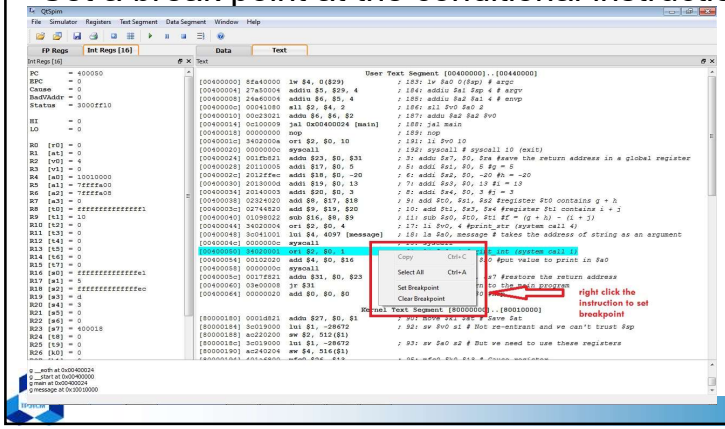
Program data



79

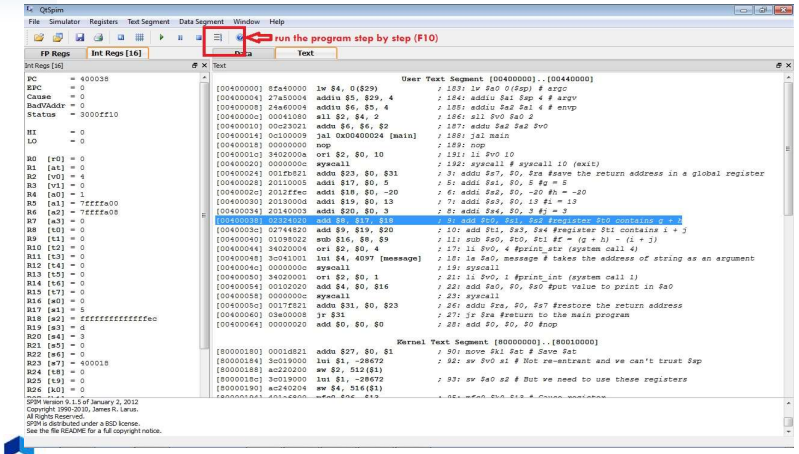
Set a break point

Set a break point at the conditional instruction



80

Debug by stepping your code line by line



81

Example

```
        .text
        .globl main
main:
        li      $t0, 0x2            # $t0 ← 0x2
        li      $t1, 0x3            # $t1 ← 0x3
        addu $t2, $t0, $t1          # $t2 ← ADD($t0, $t1)

        .text
        .globl main
main:
        ori     $t0, $0, 0x2        # $t0 ← OR(0, 0x2)
        ori     $t1, $0, 0x3        # $t1 ← OR(0, 0x3)
        addu $t2, $t0, $t1          # $t2 ← ADD($t0, $t1)
```

82

Example

```
        .data
n:      .word 0x2
m:      .word 0x3
r:      .space 4

        .text
        .globl main
main:
        lw      $t0, n              # load n to $t0
        lw      $t1, m              # load m to $t1
        addu $t2, $t0, $t1          # $t2 ← ADD($t0, $t1)
        sw      $t2, r              # store $t2 to r
```

83

Example

```
        .data
n:      .word 0x2, 0x3, 0x4

        .text
        .globl main
main:
        la $t5, n                   # load address of n to $t5
        lw $t0, 0($t5)              # load n to $t0
        lw $t1, 4($t5)              # load n+4 to $t1
        addu $t2, $t0, $t1          # $t2 ← ADD($t0, $t1)
        sw $t2, 8($t5)              # store $t2 to n+8
```

84