Chapter-2

Instructions : Language of the Computers

Instruction set: An instruction set is the collection of instructions of computers. Different computers have different instruction sets, but many aspects in common.

The MIPS Instruction set: It is developed at Stanford and commercialized by MIPS Technologies. It represents typical modern Instruction set Architecture (ISA). MIPS is widely used in consumer electronics, networking, storage, cameras and printers. It has large share of embedded core market.

Arithmetic operations:

Use 3 operands : 2 source and 1 destination

```
add a,b,c        # a = b+c
```

All arithmetic operations follow this structure for simplicity and regularity, making implementation ~~est~~ easier and higher performance at low cost.

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

MIPS code:

```
add t0, g, h
add t1, i, j
sub f, t0, t1
```

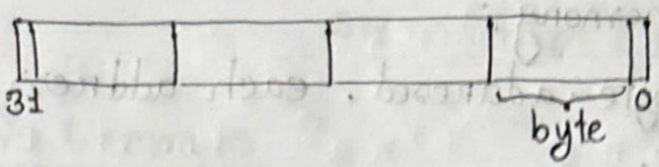
(1) Design Principle 1 : "Simplicity favours regularity"

(enables higher performance at lower cost) (makes implementation simpler)

Register Operands:

faster than memory → used for frequently accesed data

- Arithmetic instructions use register operands for efficiency
- MIPS has 32×32 bit register file (32 registers, each 32 bit)
- Registers are numbered 0 to 31 and a 32-bit value is called a word. word = 4×8bit = 32bit = 4 byte



- Book (Page-65) MIPS chart
- Book (Page-105) Chart

(2) Design Principle-2
"Smaller is faster"
because main memory has millions of locations

0	$zero	the constant value 0
2-3	$v0 - $v1	return;
4-7	$a0-$a3	function call
8-15	$t0-$t7	temporary variable
16-23	$s0-$s7	Saved
24-25	$t8-$t9	More temporaries
28	$gp	Global pointer
29	$sp	Stack "
30	$fp	Frame "
31	$ra	Return address → program counter ক save হয়

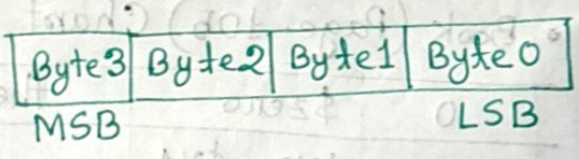
- Book last 2 Page charts

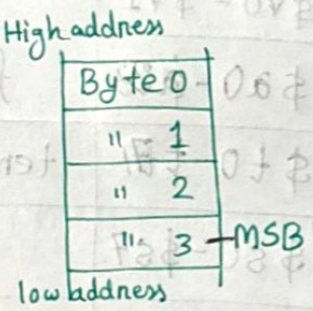
Example: f = (g+h) - (i+j) → ram ত g, h, i, j থাকে
f..., j in $s0, ..., $s1 → register ত থাকে

Memory Operands:

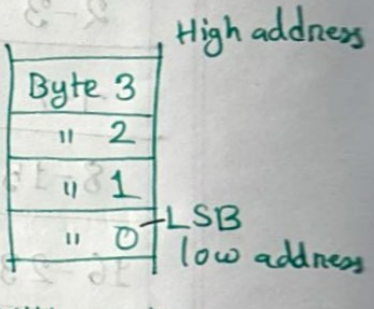
- Main memory stores composite data (arrays, structures, dynamic data).
- Arithmetic operations require load values from memory into registers and store result from register to memory.
- MIPS is byte-addressed. each address ~~inde~~ identifies an 8-bit byte.
- Words are aligned in memory.

- **Big Endian (MIPS default):** Most-significant byte at least address of a word. The MSB at address 0, the LSB at address 3.





Big Endian
MSB at least address



Little endian
LSB at least address

- **Little endian:** Least-significant byte at least address. Byte address 0 1 2 3. MSB at address 3, LSB at address 0.

Example

① $g = h + A[8]$

g → $s1, h → $s2, base address of A in $s3

MIPS Code: Index 8 requires 32 offsets

```
lw  $t0, 32($s3)   # load word
add $s1, $s2, $t0
```

offset

Example: C code: A[12] = h + A[8];

$s2 $s3

MIPS code:

```
lw   $t0, 32($s3)    # load word at memory address
add  $t0, $s2, $t0
sw   $t0, 48($s3)    # Store word
```

☐ **Register Vs Memory:**

- Registers are faster to access than memory.
- Registers can directly operate arithmetic or logical operations. Operating on memory, data requires loads and stores, which means we cannot directly operate on memory values.
- Compilers aims to keep variables in registers as much as possible. This reduce the need to access slower memory and improves performance.

☐ **Immediate Operands:** It is a constant value that is directly specified within an instruction, rather than stored in memory or register.

```
addi $s3, $s3, 4
```

add immediate

- No subtract immediate instruction: `addi $s2, $s1, -1`

(3) **Design Principle-3: "Make the common case fast"**

Small constants are common

Immediate operand avoids a load instruction.

Unsigned Binary Integers:

" " " are binary numbers that represent only non-negative values (0 and positive numbers)

Range : 0 to $2^n - 1$; 32 bit range : 0 to 4294967295

Given, n bit number,

$$X = X_{n-1}2^{n-1} + X_{n-2}2^{n-2} + \ldots + X_1 2^1 + X_0 2^0$$

Example: 0000 0000 0000 0000 0000 0000 0000 $1011_{(2)}$ ← binary

$= 0 + \ldots + (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)$

$= 0 + \ldots + 8 + 4 + 2 + 1$

$= 11_{(10)}$ → Decimal

The unsigned binary number $0000000000000000000000001011_2$ is 11 in decimal.

2s-Complement Signed Integers:

representation ← Common way to represent signed integers in binary. It allows for both positive and negative numbers

for n-bit number,

$$X = -X_{n-1}2^{n-1} + X_{n-2}2^{n-2} + \ldots + X_1 2^1 + X_0 2^0$$

Range : -2^{n-1} to $+2^{n-1} - 1$

32 bit Range: $-2,147,483,648$ to $+2,147,483,648$

Example: $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$

$= (-1 \times 2^{31}) + (1 \times 2^{30}) + (1 \times 2^{29}) + \ldots + (1 \times 2^2) + (0 \times 2^1) + (0 \times 2^0)$

$= -2147483648 + 2,147,483,644 = -4_{10}$

$2^{31} - 2^2$

In 2's complement, the most significant bit (MSB) is the sign bit which determine if a number is positive or negative:

0 → Non-negative/positive

1 → Negative number

Most negative : 1000 0000 .... 0000 → -2,147,483,648

Most positive : 0111 1111 .... 1111 → +2,147,483,647

0 : 0000 0000 0000

-1 : 1111 1111 1111

□ Signed negation: Signed negation ~~is~~ mean changing the sign of a number. turn positive into a negative number or vice versa.

- Flip all the bit (0→1, 1→0)/Complement and add 1

$X + \bar{X} = 1111 \ldots 111_2 = -1$

$\bar{X} + 1 = -X$

Example: negate +2, 0000 0000...0010

Complement: 1111 1111...1101

+1

1111 1111...1110 = -2

Sign Extension:

Process of increasing the number of bits in a signed binary number while preserve it value and sign. It helps to convert small signed number into larger format (8bit to 16bit, 16bit to 32bit etc)

In MIPS instruction set,

- addi : extend immediate value
- lb, lh : extend loaded byte / loaded halfword.
- beq, bne : extend the displacement

if (a==b) if (a!=b)

Example: 8bit to 16bit

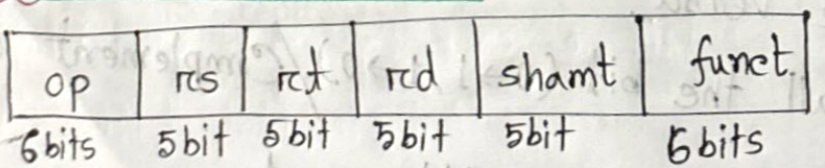
+2 : 0000 0010 ⇒ 0000 0000 0000 0010

−2 : 1111 1110 ⇒ 1111 1111 1111 1110

Instructions are encoded in binary called machine code.

MIPS (R) format Instructions:-

(register operation)



- op: operation code (opcode)
- rs: first source register number
- rt: second source register number
- rd: destination " "
- shamt: shift amount (used for shift instructions, otherwise 00000)
- funct: function code (extends opcode)

Example: add $t0, $s1, $s2

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

এই value গুলো MIPS register table এর value

$000000\ 10001\ 10010\ 01000\ 00000\ 100000_2 = 02324020_{16}$

- Hexadecimal Value: (4 bits per hex digit)

0	0000
1	0001
2	0010
3	0011

4	0100
5	0101
6	0110
7	0111

8	1000
9	1001
a	1010
b	1011

c	1100
d	1101
e	1110
f	1111

Example: eca8 6420

1110 1100 1010 1000 0110 0100 0010 0000

- MIPS (I)-format Instruction:- (used for operations that involve an immediate value, memory access (load/store)

(Immediate, Load/Store)

op	rs	rt	constant or address
6 bit	5 bit	5 bit	16 bit

- opcode
- rs: source register
- rt: Destination register
- Address: Offset added to base address in rs

  Constant: -2^{15} to $2^{15}-1$

(4) Design Principle 4: "Good design demands good compromises"

- **Stored Program Computers:** It allows computer to store both instruction and data in same memory. Instructions represented in binary, just like data. Binary compatibility allows compiled programs to work on different computers. Programs can operate on programs, compilers, linkers.

- **Logical Operations:**

Operation	C	Java	MIPS
Shift left	<<	<<	sll → Shift left logical
Shift right	>>	>>>	srl → Shift right logical
Bitwise AND	&	&	and, andi
Bitwise OR	|	|	or, ori
Bitwise NOT	~	~	nor

Useful for extracting and inserting groups of bits in a word.

- **Shift operation:**
  - Shamt: how many positions to shift
  - sll: Shift left logical
    - Shift left and fill with 0 bits
    - sll by i bits multiplies by 2^i
  - srl: Shift right logical
    - Shift right and fill with 0 bits
    - srl by i bits divided by 2^i (unsigned only)

```
sll $t0, $t1, 5
Suppose, $t1 is
0000....00000000 0000 1010
 shift left by 5,
0000....0000000101000000
```

select or modify

- **AND Operation:** Useful to mask bits in a word. [If both bits 1, result will be 1 otherwise 0.]

  and $t0, $t1, $t2

  Example:

  $t2 = 0000 0000 0000 0000 0000 1101 1100 0000

  $t1 = 0000 0000 0000 0000 0011 1100 0000 0000

  ∴ $t0 = 0000 0000 0000 0000 0000 1100 0000 0000

- **OR Operation:** Useful to include bits in a word.

  same as previous example, [If any bit is 1, result will be 1 otherwise 0]

  or $t0 $t1, $t2

  ∴ $t0 = 0000 0000 0000 0000 0011 1101 1100 0000

- **NOT Operation:** Useful to invert bits in a word. [Change 0 → 1, 1 → 0]

  Example: nor $t0, $t1, $zero → Register 0; always read as zero

  $t1 = 0000 0000 0000 0000 0011 1100 0000 0000

  ∴ $t0 = 1111 1111 1111 1111 1100 0011 1111 1111

- **Conditional Operation:** Performed using branch instructions, which allow ~~you~~ to execute different parts of the program based on conditions. (equality, greater than, less than)

  ① beq (Branch if Equal)

  beq rs, rt, L1

  if (rs == rt) branch to instruction labeled L1;

  label mark a specification location in the program code

branch if not equal

② bne rs, rt, L1

if (rs != rt) branch to instruction labeled L1;

③ j L1

unconditional jump to instruction labeled L1;

□ Compiling If statement:-

C Code: if (i==j) f = g+h;
else, f = g-h;

i stored in $s3
j " " $s4
f " " $s0
g " " $s1
h " " $s2

MIPS Code:

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

□ Compiling Loop statement:-

C Code:

```
While (save[i] == k)
    i += 1;
```

i → $s3
k → $s5
address of save → $s6

MIPS Code:

```
    sll  $t1, $s3, 2
    add  $t1, $t1, $s6
    lw   $t0, 0($t1)
    bne  $t0, $s5, Exit
    addi $s3, $s3, 1
    j    Loop
Exit:
```

Basic Blocks:

A basic block is a sequence of instruction that has
- no embedded branches (except at the end)
- no branch targets (except at the beginning)

- A compiler identifies basic blocks to optimize code execution.
- An advanced processor can accelerate execution of basic blocks.

More conditional Operations:

- Set result 1 if condition is true, otherwise set to 0.
- slt, rd, rs, rt : if (rs < rt){ rd = 1 } else { rd = 0 }

  slt → set less than
- slti, rt, rs, constant: if (rs < constant) rt = 1 else rt = 0

  slti → set less than immediate

• Using with branch Instructions:

Combination with beq, bne

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

slt → if ($s1 < $s2){ $t0 = 1 } else { $t0 = 0 }

bne → if ($t0 != 0) branch to label L

Why not blt, bge, etc?

Hardware for $>, <, \leq, \geq$ is slower than $=, \neq$. Combining with branch involves more work per instruction, requiring a slower clock. beq and bne are the common case which is a good design compromise.

Signed Vs Unsigned:-

Signed comparison slt, slti

Unsigned " sltu, sltui

Example:

$s0 = 1111 1111 1111 1111 1111 1111 1111 1111

$s1 = 0000 0000 0000 0000 0000 0000 0000 0001

- slt $t0, $s0, $s1

  ⇒ -1 < +1 ⇒ $t0 = 1

- sltu $t0, $s0, $s1

  ⇒ +4,294,967,295 > +1 ⇒ $t0 = 0

Procedure Calling:

1. Place parameters in registers
2. Transfer control to procedure
3. Acquire storage for "
4. Perform procedure's operations
5. Place result in register for caller
6. Return to place of call

Procedure call Instructions:

Procedure call: (jump and link) (jal) → address of the next instruction

(jal)

- saves return address in $ra
- Jumps to the procedure (target address)

Procedure return: jr $ra

- Copies $ra to program counter
- Can also be used for computed jumps (e.g. switch statement)

Leaf Procedure Example:

C code:

```
int leaf_example(int g,h,i,j)
{  int f;
   f = (g+h)-(i+j);
   return f;
}
```

Arguments
g, h, i, j in $a0 .... $a3
f in $s0
(need to save $s0 on stack)
Result in $v0

MIPS code: leaf example:

```
addi $sp, $sp, -4      } save $s0 on stack
sw   $s0, 0($sp)
add  $t0, $a0, $a1     } Procedure body
add  $t1, $a2, $a3
sub  $s0, $t0, $t1
add  $v0, $s0, $zero   } Result
lw   $s0, 0($sp)       } Restore $s0
addi $sp, $sp, 4
jr   $ra               } Return
```

Non-leaf Procedure:

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
  - Its return address
  - Any argument and temporaries needed after call
- Restore from the stack after call.

Example:

C code:

```
int fact(int n)
{ if (n<1)
    return f;          #[0! = 1]
  else
    return n* fact(n-1);
}
```

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                 # return
L1:   addi  $a0 , $a0, -1
      jal   fact
      lw    $a0 , 0($sp)
      lw    $ra, 4($sp)
      addi  $sp, $sp , 8
      mul   $v0, $a0, $v0
      jr    $ra
```

Character Data:

- **Byte-encoded character sets:**

  ASCII : 128 characters
  — 95 graphic, 33 control

  Latin-1 : 256 characters
  — ASCII, +96 more graphic characters

- **Unicode: 32 bit character set:** (All alphabets and symbol)
  - Used in Java, C++ wide characteristics
  - UTF-8, UTF-16

  it keeps ASCII as 8 bits uses 16 bits for other characters.

Byte / Halfword Operations:

Most operations work with 32 bit (word-sized) data, but sometimes we need to access bytes (8-bit) or halfwords (16-bit). MIPS provides special load, store instructions for these smaller data sizes.

- Load bytes (lb, lbu)

  lb rt, offset(rs)
  lbu rt, offset(rs)

- Half word operations (lh, lhu)

  lh rt, offset(rs)
  lhu rt, offset(rs)

- Store byte (sb) sb rt, offset(rs)
- Store Halfword (sh) sh rt, offset(rs)

☐ **32-bit constants:** In MIPS assembly, registers are 32-bit but there is no direct instruction to load a full 32-bit constant into a register in one step.

- lui (Load Upper Immediate)
  - Loads the upper 16 bits of a register
  - clears right 16 bits of rt to 0.
- ori (OR immediate)
  - loads the lower 16 bits

```
lui  $s0, 61
ori  $s0, $s0, 2304
```

0000 0000 0111 1101	0000 0000 0000 0000
0000 0000 0111 1101	0000 1001 0000 0000

☐ **Branch addressing:** Branch instructions use PC-relative addressing, meaning they specify an offset relative to the program counter (PC) rather than an absolute address / Target address.

~~Most~~ • Branch instructions specify opcode, two registers, target address (16 bits) — 6 bit — (rs, rt)

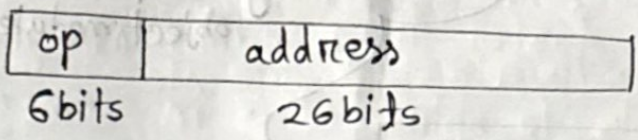
- Most branch targets are near branch forward or backward

Can jump forward and backward within limited range

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

- PC related addressing
  - Target address = PC + (offset × 4)
  - PC already incremented by 4 when the branch instruction is executed.

□ **Jump Addressing:** Jump ~~addressin~~ Instruction (j and jal) use absolute addressing, meaning they can jump anywhere in the text segment.



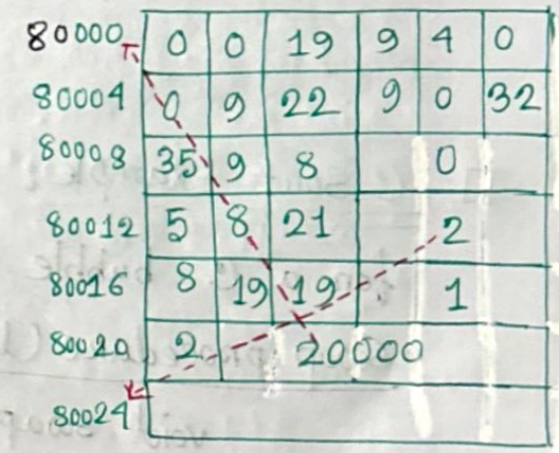
(Pseudo) Direct Jump addressing

- Target address = $PC_{31...28}$: (address × 4)

MIPS encodes only the lower 26 bits of the target address and reconstructs the full address when executing the jump.

Example: Assume loop at location 80,000

```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    loop
Exit:
```



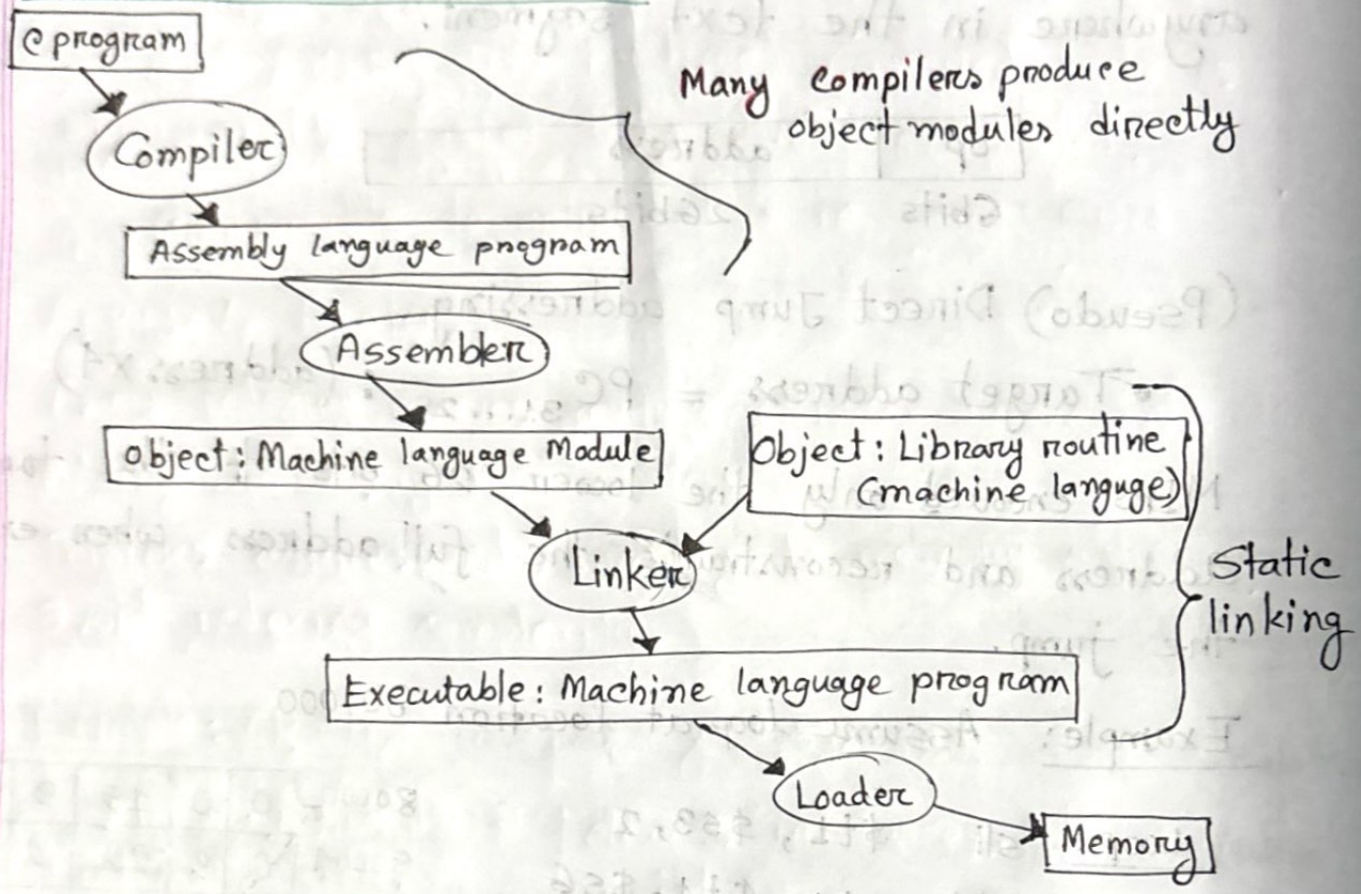
□ **Branching Far away:** When a branch target is too far away to be encode with 16-bit offset, the assembler can rewrite the code handle the long distance brance

```
      beq  $s0, $s1, L1
      bne  $s0, $s1, L2
      j    L1
L2:
```

} combination of branch and jump instruction

Page-53 (Addressing Mode Summary)

Translation and startup:



C sort Example: Illustrates use of assembly instructions for a C bubble sort function.

- **Swap procedure (leaf):**

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

v in $a0, k in $a1, temp in $t0

```
sll $t1, $a1, 2
add $t1, $a0, $t1
lw  $t0, 0($t1)
lw  $t2, 4($t1)
sw  $t2, 0($t1)
sw  $t0, 4($t1)
jr  $ra
```

- Non-leaf (calls swap):

C code:
```
void sort (int v[], int n)
{ int i, j;
  for(i = 0; i<n; i+=1) {
    for(j = i-1; j >=0 && v[j] >v[j+1]; j-=1) {
      swap(v,j);
    }
  }
}
```

Argument n in $a0
Result in $v0

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               # return.
```

Lessons learnt:

- Instruction count and CPI are not good performance indicator in isolation.
- Compiler optimizations are sensitive to the algorithm.
- JAVA / JIT (Java in time) compiled code is significantly faster than JVM interpreted.
  - Comparable to optimized C in some cases.

□ Arrays Vs Pointers:

Array indexing involves multiplying index by element size, Adding to array base address.

Pointers correspond directly to memory addresses, can avoid indexing complexity.

Example: Clearing and Array (Page-62)

```
clear1(int array[],int size) {
  int i;
  for(i=0; i<size;i+=1)
  array[i]=0;
}
```

```
loop: move $t0,$zero   #i=0
      sll  $t1,$t0,2   #$t1=i*4
      add  $t2,$a0,$t1

      sw   $zero,0($t2) #array[i]=0
      addi $t0,$t0,1   #i=i+1
      slt  $t3,$t0,$a1

      bne  $t3,$zero,loop1
                       #goto loop1
```

```
clear2 (int *array, int size){
  int *p
  for (p=&array[0];p<&array[size];
                          p=p+1)
    *p=0;
}
```

```
      move $t0,$a0     #p=&array[0]
      sll  $t1,$a1,2   #$t1=size*4
      add  $t2,$a0,$t1 #t2=&array[size]

loop2: sw  $zero,0($t0)#Memory[p]=0
      addi $t0,$t0,4   #p=p+4
      slt  $t3,$t0,$t2

      bne  $t3,$zero,loop2
                       #goto loop2
```

□ **Comparison of Array Vs Pointer:** When dealing with loops both arrays and pointers can be used for iteration. However, pointers are sometimes preffered for optimization due to strength reduction and Induction variable elimination. Using pointers can make code harder to read and error-prone (e.g. pointer arithmatic mistake, dangling pointers). Better to make program clearer and safer.

- **Induction variable:** A variable that gets increased or decreased by a fixed amount of every iteration of a loop.

□ **ARM & MIPS Similarities:**

ARM is the most popular embedded core.

	ARM	MIPS
Date announced	1985	1985
Instruction size	32 bits	32 bits
Address space	32-bit flat	32-bit flat
Data alignment	Aligned	Aligned
Data addressing modes	9	3
Registers	15 x 32 bit	31 x 32-bit
Input/Output	Memory mapped	Memory mapped

□ Compare and branch in ARM:

ARM uses condition codes (like flags) to track results of arithmetic and logical operations. These incude negative, zero, carry, overflow. Some instructions only set condition codes without storing result. This helps in making decisions efficiently.

- Each ARM instruction can be conditional.
- The top 4 bits of an instruction decide if it should run or not.
- Can avoid branches over single instructions.

□ Instruction encoding: Pg-66

ARM uses more complex encoding but allows conditional execution. MIPS is simpler and more uniform, making decoding easier.

□ Fallacies: Powerful instruction means higher performance

- Fewer instructions ~~ree~~ seem good, but complex instructions take longer to execute. this can slow everything down.

Solution: Use simple instructions and let compilers optimize them.

"Assembly code is the best for high performance."

Modern compilers are ~~comp~~ optimized for processors and do a better job than humans. in most cases. More code ⟶ more bugs and less productivity.

Pitfalls:

- Word-aligned memory → Increment by 4, not by 1.
- Automatic variable → Avoid returning pointers to local (stack) variables.
- Pointers & Function Calls → Be careful when passing pointer to local variable, they disappear after function returns.

Conclusion:

- Design Principles:
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Make the common case fast
  4. Good design demands good compromises
- Layers of software/hardware:

  Compilers assembler, hardware
- Two major approaches to design a CPU,
  1. RISC (Reduced Instruction Set Computing): Keep instruction set small and simple so that the processor can execute instructions faster.
  2. CISC (Complex " " "): Use fewer, more complex instructions that can do multiple operations in one step.