CSE 305
Computer Architecture

Instructions

Prepared by

Madhusudan Basak

Assistant Professor

CSE, BUET

*Some modifications done by Saem Hasan

Instructions

- A computer is run by instructions
- Instruction Set
  - All possible instructions for a CPU

Instruction Set Architecture

- Instruction set architecture is the attributes of a computing system as seen by the assembly language programmer or compiler.
  - Instruction Set (what operations can be performed?)
  - Instruction Format (how are instructions specified?)
  - Data storage (where is data located?)
  - Addressing Modes (how is data accessed?)
  - Exceptional Conditions (what happens if something goes wrong?)

Design Philosophy

It is easy to see by formal-logical methods that there exist certain [instruction sets] that are in abstract adequate to control and cause the execution of any sequence of operations. ... The really decisive considerations from the present point of view, in selecting an [instruction set], are more of a practical nature: simplicity of the equipment demanded by the [instruction set], and the clarity of its application to the actually important problems together with the speed of its handling of those problems.

Burks, Goldstine, and von Neumann, 1946

MIPS Architecture

- Acronym of "**M**icroprocessor without **I**nterlocked **P**ipelined **S**tages"
  - Is a RISC (Reduced Instruction Set Computer) ISA (Instruction Set Architecture)
  - Developed by MIPS Technologies (then MIPS Computer Systems) in 1985

xm
এ
থাকবে

কোনো decision নিলে যেই কোন design principle follow করে।

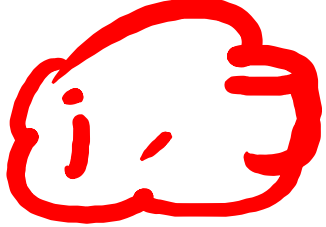
Design Principles

- *Design Principle 1:* Simplicity favors regularity.
- *Design Principle 2:* Smaller is faster.
- *Design Principle 3:* Good design demands good compromises.

Simplicity favors regularity

- The instructions of MIPS are fixed and rigid
- Rigidity ensures Regularity
- Simplicity favors regularity

R-format



Example Instruction

- *Add* and *subtract* instructions always follow the following fixed format

add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three register operands
subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3	Three register operands

register (destination)

subtraction হবে এই২টোর উপর

$$a = b + c + d + e$$

```
add a, b, c     # The sum of b and c is placed in a
add a, a, d     # The sum of b, c, and d is now in a
add a, a, e     # The sum of b, c, d, and e is now in a
```

Example Instruction

- *Add* and *subtract* instructions always follow the following fixed format

add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three register operands
subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3	Three register operands

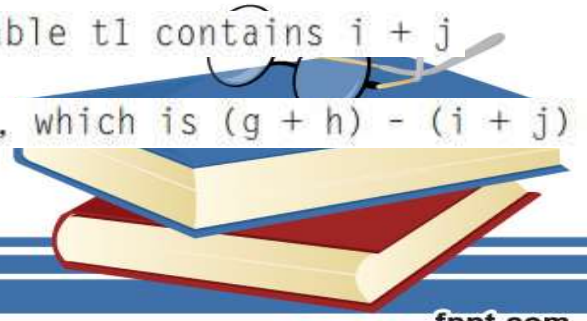
```
a = b + c;
d = a - e;
```

```
add a, b, c
sub d, a, e
```

or

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

```
add t0,g,h # temporary variable t0 contains g + h
add t1,i,j # temporary variable t1 contains i + j
sub f,t0,t1 # f gets t0 - t1, which is (g + h) - (i + j)
```



MIPS Instruction Properties

- Arithmetic operations only takes register values as operands
- Size of a register : 32 bit
- Number of registers: 32 → 2^5 (প্রতিটা register representation এ 5 bit লাগবে)
  - Satisfies *Design Principle 2:* Smaller is faster.
    - Large number of registers -> longer time for electronic signal to travel -> Increased clock cycle
      ↳ access time বেশি লাগবে।
    - Large number of registers -> Increased number of control bits

- A number is dedicated to a particular register.
  - 5 bit are reserved to indicate each register when the count is 32
  - Compiler maps a program variable to a register
  - There is a number assigned against each register

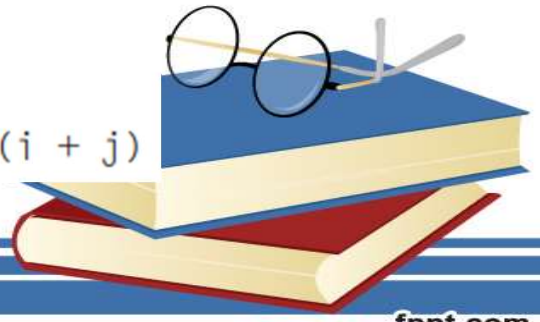
MIPS Instruction Properties

- Example

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

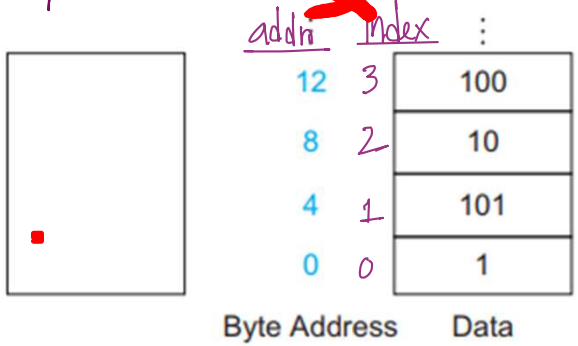
```
add t0,g,h # temporary variable t0 contains g + h
add t1,i,j # temporary variable t1 contains i + j
sub f,t0,t1 # f gets t0 - t1, which is (g + h) - (i + j)
```

```
add $t0,$s1,$s2 # register $t0 contains g + h
add $t1,$s3,$s4 # register $t1 contains i + j
sub $s0,$t0,$t1 # f gets $t0 - $t1, which is (g + h)-(i + j)
```



MIPS Instruction Properties

- What about data structures (Arrays and Structures)?
  - These are kept in memory and a particular element is transferred to registers whenever required
- Compiler
  - places an array/structure into memory
  - keeps the starting address of the memory in a register
  - tracks which register is used for which array/structure



- Data Transfer Instructions
  - load word: *lw*
  - store word: *sw*

integer: *(arr + 4xi) → offset. Base element থেকে কত দূরে আছে।

char: 1 byte করে add হবে।

সব operation register এ হয়।

Array থাকে memory তে।

$A[2] = A[2] + 5$

A এর starting address register

$s3 তে থাকে

base register – memory-র starting addr. রাখে।

A[2] → offset হবে 8.

$s3 8

to তে load →

$t0 , 8 ($s3)

lw $reg , offset($base reg)

sw $reg , offset ($base reg)

২টা operation

i) load word – memory-র addr থেকে register এ আনি

ii) store word – register থেকে memory তে রেখে আসবে।

Data Transfer Instructions

- Example
- $A[12] = h + A[8]$

h → $s2

A 의 base register $s3

i) A[8] load to t0

$t0, 32($s3)

8 × 4

```
lw   $t0,32($s3)  # Temporary reg $t0 gets A[8]
add  $t0,$s2,$t0  # Temporary reg $t0 gets h + A[8]
sw   $t0,48($s3)  # Stores h + A[8] back into A[12]
```

4 × 12

add $dest_reg, $source_reg1, $source_reg2

যখন 32 থেকে বেশি variable থাকে তখন যেগুলো বেশি use হয় সেগুলো register এ map করে রাখে। বাকিগুলো memory তে রাখে।

Spilling Registers

xm: 40টি variable থাকলে MIPS কিভাবে handle করবে?

- What if the number of variables and data structures is more than 32?
  - The process of putting less commonly used variables into memory is called Spilling Registers.
  - Keeps the most frequently used variables in registers and less frequents one in memory
  - Moves variables around registers and memory

Operation with Constants

- Add *immediate* or *addi* instruction
  - Takes one register and one constant as input

```
addi    $s3,$s3,4      ⟶    # $s3 = $s3 + 4
```

add immediate

register এর মধ্যে যখন value add করা লাগবে। ⟶

addi
ori
subi

immediate operation

- Constant 0 is used very often (e.g., for transfer operations)
  - A register $zero is dedicated for this purpose

```
add    $s2, $s3, $zero        #$s2=$s3
```

machine code এ 5 bit এ convert হয়।

a=b
mov a, b } assembly তে। MIPS এ add দিয়ে করা লাগে।

- Is an instruction represented using register names?

register এ কাজগুলো তাড়াতাড়ি হয়।

add, sub এগুলো কিভাবে? — instruction গুলোও number এ convert হয়।

machine code — memory তে থাকে।

শুধু data থাকবে এমন না।

Representing Instructions in the Computer

- Key principles of Today's computer instructions
  - Instructions are represented as numbers
  - Program are stored in memory to be read or written, just like data

- instruction memory → machine code convert হয়ে load হয়।
- data memory → data থাকে।

Representing Instructions in the Computer

- An instruction is a sequence of bits
- Each register is mapped to a number
  - There is a convention to map register names into numbers (e.g., registers $s0 to $s7 map onto registers 16 to 23) → xm এ দিয়ে দেওয়া থাকবে।

- Each instruction is 32 bit long (size of a word) → <32 হলে register এ bit খালি থাকে >32 হলে বেশি register লাগে।
  - Satisfies *Design Principle 1:* Simplicity favors regularity.

- Instruction Format: format used by an instruction

question – 32bit কাখলাম। কোন design principle follow করে?

An MIPS Instruction Format

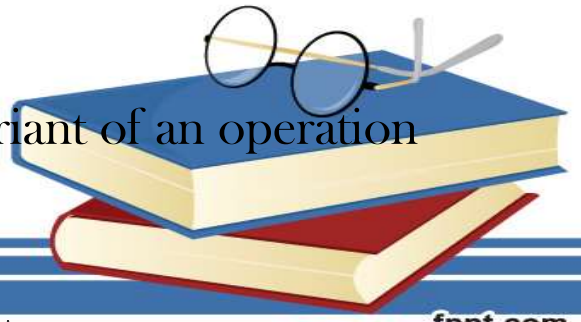
- R-Format → add, sub, or, and, NOR (যেগুলোতে তিনটাই register লাগে)
  - Deals with arithmetic operations with registers

op code

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

  - *op*: shorthand of opcode, denotes operation type and format type, কোন function করবে।
  - *rs*: Source Register1
  - *rt*: Source Register 2
  - *rd*: Destination Register
  - *shamt*: Shorthand of shift amount
  - *funct*: shorthand of function code, denotes the specific variant of an operation

add $t0, $t1, $t2
funct rd rs rt



op code — 000000 (always)
R format বোঝায়

অন্য format এ funct নামে extra bit নাই। তখন op code দিয়ে operation type ও বুঝাবে।

An MIPS Instruction Format

$t0-8
$s1-17
$s2-18

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

add $t0,$s1,$s2

add এ shift লাগে না। তাই shamt = 0

0	17	18	8	0	32

একটা instruction

000000	10001	10010	01000	00000	100000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

R-format

6 bit করেও রাখতে পারি। কিন্তু তখন funct এর জন্য bit বলে যাবে। কয় সংখ্যক operation করা যাবে।

Would it be good to use just one format for all kind of operations?

No!

Design Principle 3: Good design demands good compromises.


```
add  $t0, $t1, 5
lw   $t0, 32($s3)
```

} R-format এ করা যাবে না।

total 32 bit, first 6 bit read → কোন format
then 5 bit read → 1st register এর value ।

Another MIPS Instruction Format

- I-Format — addi, ori, subi, lw, sw
  - Deals with immediate and data transfer operations

destination register

  - *rt*: Here *rt* represents the destination register

addi $t0, $t1, 5 (২টা register)

funct বলে আলাদা bit নাই। op code দিয়েই format এবং operation type বোঝাবে।

MIPS Instruction Encoding (Simplified)

Instruction	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32_{ten}	n.a.
sub (subtract)	R	0	reg	reg	reg	0	34_{ten}	n.a.
add immediate	I	8_{ten}	reg	reg	n.a.	n.a.	n.a.	constant
lw (load word)	I	35_{ten}	reg	reg	n.a.	n.a.	n.a.	address
sw (store word)	I	43_{ten}	reg	reg	n.a.	n.a.	n.a.	address

R formate এ op-code always zero.

$t0 = 8
$t1 = 9
$s2 = 18

```
A[300] = h + A[300];

lw   $t0,1200($t1) # Temporary reg $t0 gets A[300]
add  $t0,$s2,$t0   # Temporary reg $t0 gets h + A[300]
sw   $t0,1200($t1) # Stores h + A[300] back into A[300]
```

source

Op	rs	rt	rd	address/ shamt	funct
35	9 t1	8 t0	1200 (offset)		
0	18 s2	8 t0	8 t0	0	32
43	9 t1	8 t0	1200 (offset)		

100011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000 0100 1011 0000		

xml এ Mips এর operation থাকলে sw থাকবেই

Summary

$s1 = 17 , $s2 = 18 , $s3 = 19

MIPS machine language

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add $s1,$s2,$s3
sub	R	0	18	19	17	0	34	sub $s1,$s2,$s3
addi	I	8	18	17	100			addi $s1,$s2,100
lw	I	35	18	17	100			lw $s1,100($s2)
sw	I	43	18	17	100			sw $s1,100($s2)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer format

Logical Operations

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Logical	and	and $s1,$s2,$s3	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	or $s1,$s2,$s3	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	nor $s1,$s2,$s3	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	andi $s1,$s2,20	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	ori $s1,$s2,20	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
R format	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl $s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant

.xml ৳

No NOT Operation?

↳ NOR দিয়েই করা যায়।

$s3 → zero register করে দিবো।

sll $s1, $s2, 10

register constant

Shift Left

- sll $t2,$s0,4 # reg $t2 = reg $s0 << 4 bits

↳ shift amount ও যাবে।

op	rs	rt	rd	shamt	funct
0	0	16	10	4	0

shift operation
হচ্ছে বোঝায়

Branching Operations

a+b
c+d
go to L1
⋮
L1 : d+e

jump

if-else

branch equal

```
beq register1, register2, L1
```

L1 → Label

```
bne register1, register2, L1
```

branch not equal

L1 → Label (assembly তে label দিয়ে কাজ হয়)

```
if (i==j)
    f = g + h;
else
    f = g - h;
```

f = g + h; → add $s0, $s1, $s2

sub $s0, $s1, $s2

MIPS Assembly Code:

```
bne $s3, $s4, Else      #go to Else if i ≠ j
add $s0, $s1, $s2       #f = g + h
j Exit
Else: sub $s0, $s1, $s2 #f = g - h
Exit:
```

mention করে দিবো
i = $s3
j = $s4
f = $s0
g = $s1
h = $s2

```
slt   $t0, $s3, $s4   # $t0 = 1 if $s3 < $s4
slti  $t0,$s2,10      # $t0 = 1 if $s2 < 10
sltu  $t1, $s0, $s1 # unsigned comparison
```

কোনো branching হচ্ছে না।

Why not *blt*?

slt → set less than.

design principle: Smaller is faster

blt → এমন কোনো operation নেই। ২ step এ করি—
1. slt দিয়ে check
২. beq দিয়ে $zero এর সাথে check

```
if (i==j)
    f = g + h;
else
    f = g - h;
```

→ add $s0, $s1, $s2

↳ sub $s0, $s1, $s2

```
bne   $s3, $s4, L1
add   $s0, $s1, $s2
j     Exit
L1: sub   $s0, $s1, $s2
Exit:
```

→ i, j equal नही
रहने L1 पे jump.

unconditional jump

or.

```
beq   $s3, $s4, L1
sub   $s0, $s1, $s2
j Exit
L1: add $s0, $s1, $s2
Exit:
```

```
if (i < j)
    f = g+h
else
    f = g-h
```

```
slt   $t0, $s3, $s4
beq   $t0, $zero, L1
add   $s0, $s1, $s2
j Exit
L1: sub   $s0, $s1, $s2
Exit:
```

Branching Operations

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Conditional branch	branch on equal	`beq $s1,$s2,25`	if (`$s1 == $s2`) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1,$s2,25`	if (`$s1 != $s2`) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1,$s2,$s3`	if (`$s2 < $s3`) `$s1 = 1`; else `$s1 = 0`	Compare less than; for `beq`, `bne`
	set on less than unsigned	`sltu $s1,$s2,$s3`	if (`$s2 < $s3`) `$s1 = 1`; else `$s1 = 0`	Compare less than unsigned
	set less than immediate	`slti $s1,$s2,20`	if (`$s2 < 20`) `$s1 = 1`; else `$s1 = 0`	Compare less than constant
	set less than immediate unsigned	`sltiu $s1,$s2,20`	if (`$s2 < 20`) `$s1 = 1`; else `$s1 = 0`	Compare less than constant unsigned
Unconditional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to `$ra`	For switch, procedure return
	jump and link	`jal 2500`	`$ra` = PC + 4; go to 10000	For procedure call

branch operation এ constant থাকে

Instruction	Example	Meaning	Comments
branch on equal	beq $s1,$s2,25	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
branch on not equal	bne $s1,$s2,25	if ($s1!= $s2) go to PC + 4 + 100	Not equal test; PC-relative
set on less than	slt $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for beq, bne

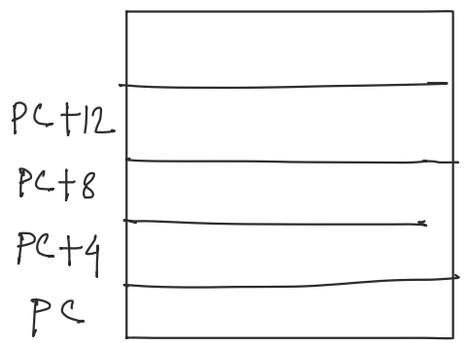
কত instruction পরে jump করা লাগবে

per instruction 4 byte.

∴ 25*4 → 100byte পরে jump.

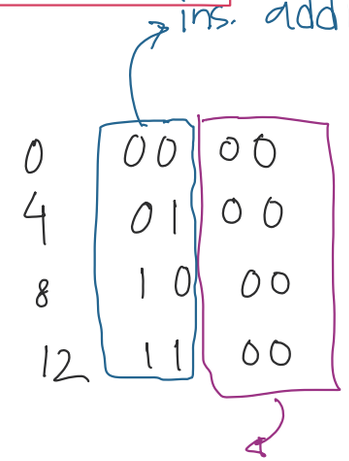
PC — program counter (instruction এখানে load হয়)

by default pc+4 করা লাগে।



jump compute করবো pc+4 এর আগেরে। কারণ pc+4 by default করাই থাকে।

xm এ আসবে



যব instruction এর address এ এই part থাকবে। তাই এই 2 bit address এ দেই না। ফলে jump এর range বাড়বে।

beq $s3, $s4, 1 →

```
if (i==j)
    f = g + h;
else
    f = g - h;
```

pc

pc+4

1 byte দুরে

pc+8

j 2500 → instruction এর address

direct jump

jr $ra (return address)

→ function থেকে return এর সময় call হয়। ra register এ direct jump. function এর কাজ শেষে কোথায় back করবে ra তে থাকে।

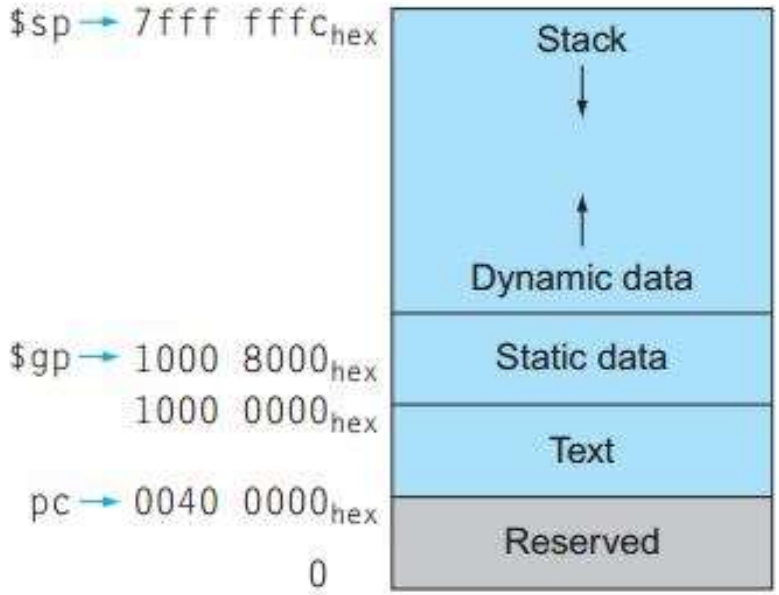
jal 2500 → $ra = pc + 4 (return করা লাগবে next instruction এ)

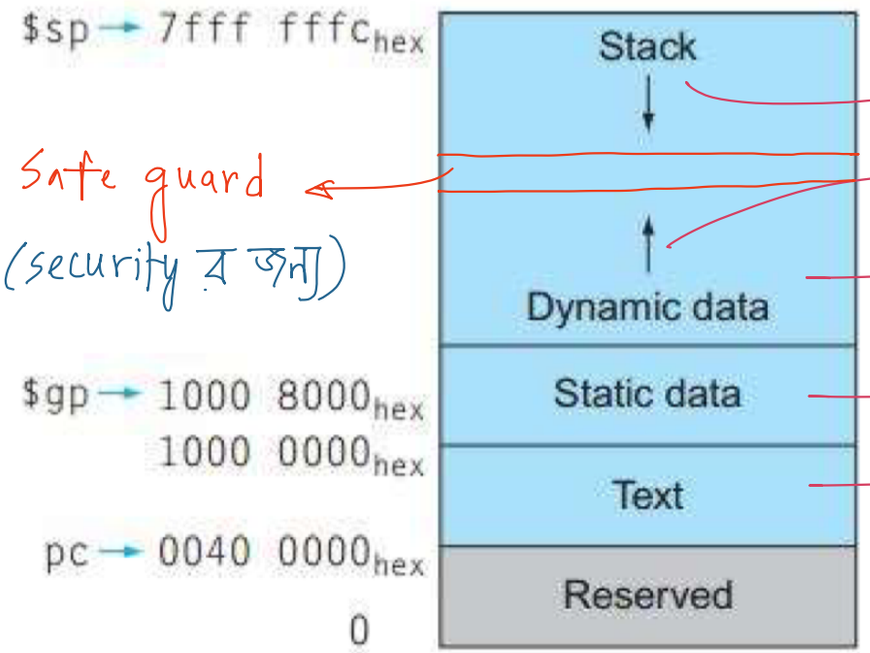
jump & link

→ 10000 এ যাবে।

Memory Allocation for Program and Data

- Reserved Memory
- Text Segment: Program Code
- Static Data Segment:
  - Contains static variables, constants, arrays
- Dynamic Data Segment (Heap):
  - Contains linked lists, variable length data
- Stack
  - Contains function local variables, parameters





higher থেকে lower address এ যায়।

lower থেকে higher address এ

Linked List, ..

static variable

Code

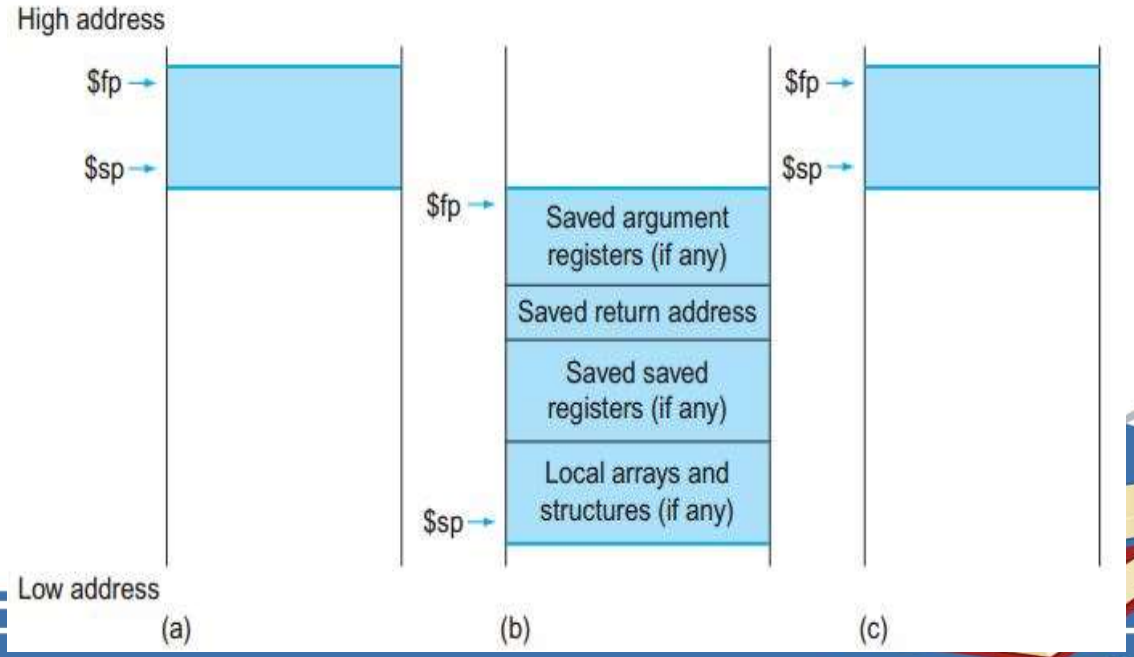
stack → function এর instruction থাকে eg. কোথায় return.

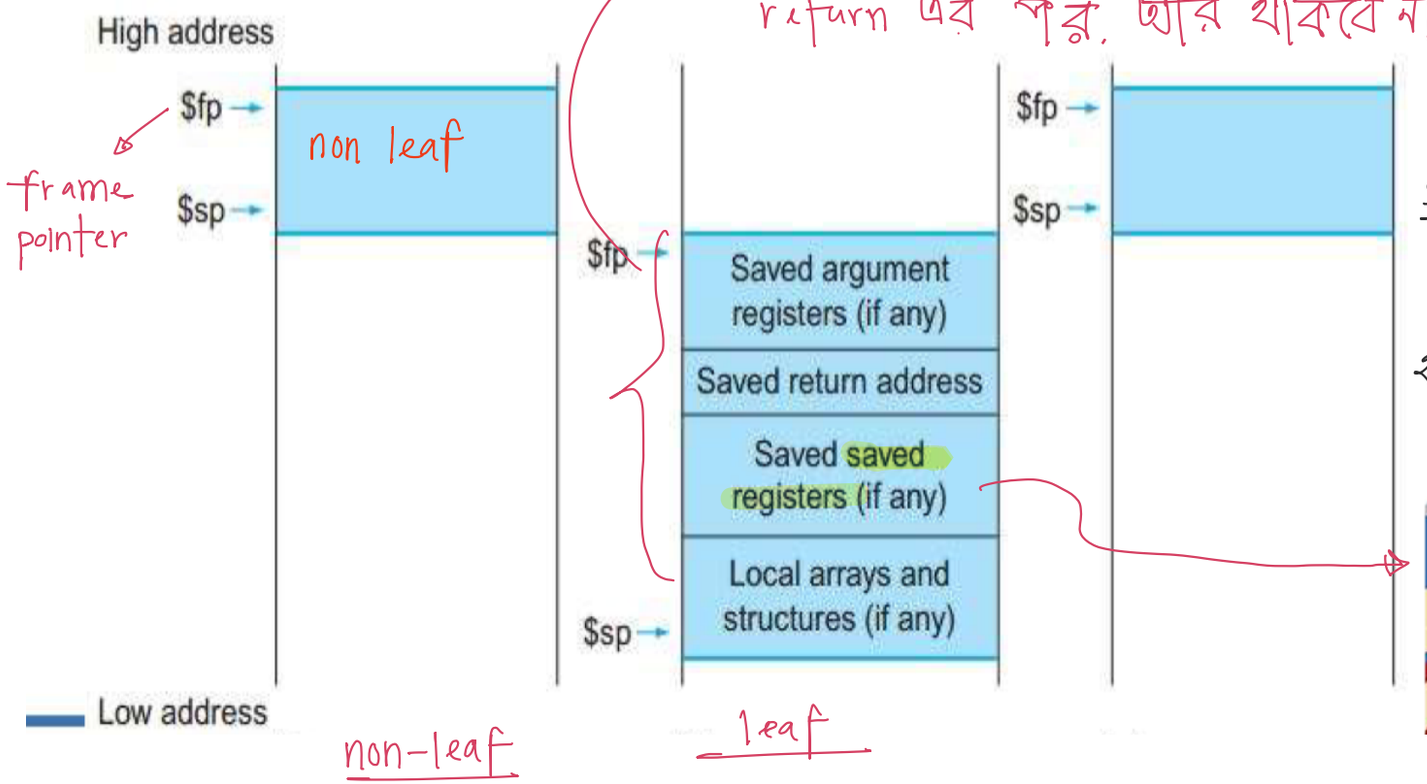
Stack During Procedure Call

- Stores/saves the values of registers and Restores those later
- Either $sp and $fp combination, or only $sp is used
- The allocated stack area by a procedure is known as activation record or procedure frame

```
int non_leaf()
{
    int a=10, b=5, c=2, d=7, x;
    x=    leaf(a,b,c,d);
    return x;
}

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





non-leaf থেকে leaf এ যাওয়ার সময়। leaf থেকে return এর পর আর থাকবে না।

non-leaf থেকে leaf func. call →

step 1. non leaf func এর argument stack এ store.

২. non leaf function কোথায় return করে সেটাও store.

3. এক func থেকে অন্য func এ যাবো then return এর পর আগের func. এর সব যেন saved থাকে।

fp → certain function এর জন্য (stack এর কোথায় আছে)

↳ starting address

sp → stack pointer (currently কোথায় আছে)

fp to sp → activation range

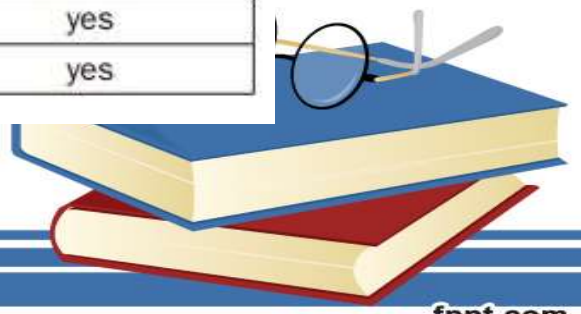
$V0 - $V1 → return value store
আরো বেশি value return করতে হলে stack use.

Stack During Procedure Call

- Stores the values of registers → tmp register ও store করা লাগতে পারে।

Name	Register number	Usage	Preserved on call?
$zero	0	The constant value 0	n.a.
$v0–$v1	2–3	Values for results and expression evaluation	no
$a0–$a3	4–7	Arguments (parameter passing)	no
$t0–$t7	8–15	Temporaries	no
$s0–$s7	16–23	Saved	yes
$t8–$t9	24–25	More temporaries	no
$gp	28	Global pointer	yes
$sp	29	Stack pointer	yes
$fp	30	Frame pointer	yes
$ra	31	Return address	yes

preserve করাই লাগবে



Procedures in MIPS

- Six steps during a procedure call
  1. Put parameters in a place where the procedure can access them.
  2. Transfer control to the procedure.
  3. Acquire the storage resources needed for the procedure.
  4. Perform the desired task.
  5. Put the result value in a place where the calling program can access it.
  6. Return control to the point of origin, since a procedure can be called from several points in a program.

Procedures in MIPS

- Six steps during a procedure call
  1. Put parameters in a place where the procedure can access them.
  2. Transfer control to the procedure.
  3. Acquire the storage resources needed for the procedure.
  4. Perform the desired task.
  5. Put the result value in a place where the calling program can access it.
  6. Return control to the point of origin, since a procedure can be called from several points in a program.

$a_0 - a_3$
আরো বেশি লাগলে
register এ দিবো।

- Place for parameters:
  - $a0–$a3: four argument registers in which to pass parameters
  - Memory (e.g., stack) can be used if more values to be passed

Procedures in MIPS

- Six steps during a procedure call
  1. Put parameters in a place where the procedure can access them.
  2. Transfer control to the procedure.
  3. Acquire the storage resources needed for the procedure.
  4. Perform the desired task.
  5. Put the result value in a place where the calling program can access it.
  6. Return control to the point of origin, since a procedure can be called from several points in a program.
- Transfer Control:
  - Jump-and-link instruction (jal)

    jal ProcedureAddress → $ra 에 pc+4 store 해줌
  - Keeps the return address for the procedure in $ra and jumps to the *ProcedureAddress*

Procedures in MIPS

- Six steps during a procedure call
  1. Put parameters in a place where the procedure can access them.
  2. Transfer control to the procedure.
  3. Acquire the storage resources needed for the procedure.
  4. Perform the desired task.
  5. Put the result value in a place where the calling program can access it.
  6. Return control to the point of origin, since a procedure can be called from several points in a program.
- Acquiring the storage resources
  - Different register values (manipulated by caller) are stored in the stack
  - Prepares the registers for operations
  - Can use the memory for data storage

Procedures in MIPS

- Six steps during a procedure call
  1. Put parameters in a place where the procedure can access them.
  2. Transfer control to the procedure.
  3. Acquire the storage resources needed for the procedure.
  4. Perform the desired task.
  5. Put the result value in a place where the calling program can access it.
  6. Return control to the point of origin, since a procedure can be called from several points in a program.

Name	Register number	Usage	Preserved on call?
$zero	0	The constant value 0	n.a.
$v0–$v1	2–3	Values for results and expression evaluation	no
$a0–$a3	4–7	Arguments	no
$t0–$t7	8–15	Temporaries	no
$s0–$s7	16–23	Saved	yes
$t8–$t9	24–25	More temporaries	no
$gp	28	Global pointer	yes
$sp	29	Stack pointer	yes
$fp	30	Frame pointer	yes
$ra	31	Return address	yes

Procedures in MIPS

- Six steps during a procedure call
  1. Put parameters in a place where the procedure can access them.
  2. Transfer control to the procedure.
  3. Acquire the storage resources needed for the procedure.
  4. Perform the desired task.
  5. Put the result value in a place where the calling program can access it.
  6. Return control to the point of origin, since a procedure can be called from several points in a program.
- Performing Tasks
  - Can perform all the operations allowed by the MIPS instruction set

Procedures in MIPS

- Six steps during a procedure call
  1. Put parameters in a place where the procedure can access them.
  2. Transfer control to the procedure.
  3. Acquire the storage resources needed for the procedure.
  4. Perform the desired task.
  5. Put the result value in a place where the calling program can access it.
  6. Return control to the point of origin, since a procedure can be called from several points in a program.
- Result storing by the callee
  - $v0-$v1: two value registers in which to return values
  - Memory (e.g., stack) can be used if more values to be returned

첫번째 return $V0
두번째 return $V1

Procedures in MIPS

- Six steps during a procedure call
  1. Put parameters in a place where the procedure can access them.
  2. Transfer control to the procedure.
  3. Acquire the storage resources needed for the procedure.
  4. Perform the desired task.
  5. Put the result value in a place where the calling program can access it.
  6. Return control to the point of origin, since a procedure can be called from several points in a program.
- Procedure returned
  - Stack is adjusted using $sp and $fp pointers
  - An unconditional jump to the address from where the caller will resume execution

$$jr \quad \$ra$$

Online

C code to MIPS

specialized register. { $a_0 - a_3$ → parameter, $v_0 - v_1$ → return

↳ normal register এর মত use করা যাবে না।

An Example

```
int non_leaf()
{
    int a, b, c, d, x;
    ...
    x=leaf(a,b,c,d);
    ...
}

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

```
non_leaf:
    ...
    add $a0, $s1, $zero      # 1. a0–a3 তে argument দেয়া।
    add $a1, $s2, $zero
    add $a2, $s3, $zero
    add $a3, $s4, $zero
    jal leaf                 # 2. function call
    add $s1, $zero, $v0      # যায়
    ...
leaf: addi $sp, $sp, -12     # adjust stack to make room for 3 items
    sw $t1, 8($sp)           # 3. stack এ store
    sw $t0, 4($sp)           # save register $t0 for use afterwards
    sw $s0, 0($sp)           # save register $s0 for use afterwards
    add $t0,$a0,$a1          # 4. function এর কাজ করা।
    add $t1,$a2,$a3          # register $t1 contains i + j
    sub $s0,$t0,$t1          # f = $t0 – $t1, which is (g + h)–(i + j)
    add $v0,$s0,$zero        # Set return Value
    lw $s0, 0($sp)           # 5. Return
    lw $t0, 4($sp)           # restore register $t0 for caller
    lw $t1, 8($sp)           # restore register $t1 for caller
    addi $sp,$sp,12          # 6. stack থেকে আগের value read করে আনলাম।
    jr $ra                   # jump back to calling routine
```

Put parameters in a place where the procedure can access them.

Transfer control to the procedure.

Acquire the storage resources needed for the procedure

Perform the desired task.

Put the result value in a place where the calling program can access it.

Return control to the point of origin, since a procedure can be called from several points in a program.

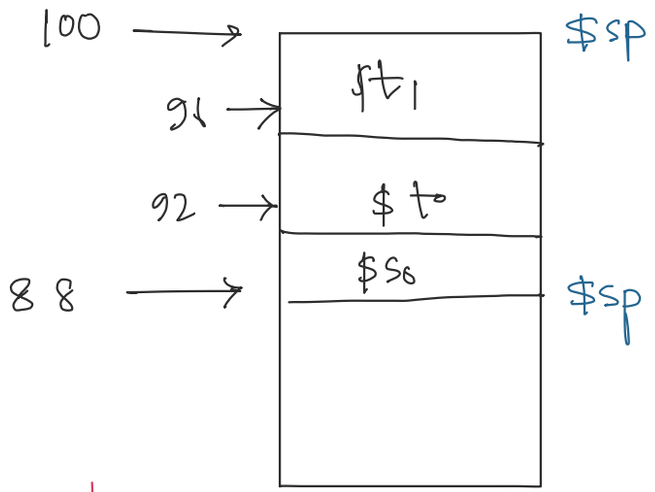
stack থেকে read করা শেষে আবার $sp কে আগের address এ set করে দিবো।

$$t = (g+h) - (i+j)$$

3 টা register use করবো। এদের value store করা লাগবে।

12 byte write করা লাগবে।

value insert → higher থেকে lower address এ যাই।



$sp 100 তে ছিলো। 3টা register save করার জন্য (100-12)/88 তে $sp কে আনলাম। Then $sp base address এর সাথে offset add করে value গুলো store করবো।

- non-leaf function এর কাজ করার সময় $t0, $t1 এর value change হয়ে যায়। তাই আগের value store করা লাগে (stack এ)।

- যে order এ store করবো তার reverse order এ load করবো।
  ↳ lower থেকে higher

t.f: কত optimized ভাবে register use করা যায়?

↳ প্রথমে function এর কাজ করে ফেলবো। then store load এর কাজ।
(Code এ)

xm এ থাকবে।

Recursion → return address বারবার change হয়।

Nested Call

```
int fact(int n)
{
    if(n<1)
        return 1;
    else
        return n*fact(n-1);
}
```

```
fact: addi $sp, $sp, -8      # adjust stack for 2 items
      sw $ra, 4($sp)         # save the return address
      sw $a0, 0($sp)         # save the argument n
      slti $t0,$a0,1         # test for n < 1
      beq $t0,$zero,L1       # if n >= 1, go to L1
      addi $v0,$zero,1       # Set the return value
      addi $sp,$sp,8         # pop 2 items off stack
      jr $ra                 # return to caller
L1:   addi $a0,$a0,-1        # n >= 1: argument gets (n - 1)
      jal fact               # call fact with (n -1)
      lw $a0, 0($sp)         # return from jal: restore argument n
      lw $ra, 4($sp)         # restore the return address
      addi $sp, $sp, 8       # adjust stack pointer to pop 2 items
      mul $v0,$a0,$v0        # return n * fact (n - 1)
      jr $ra                 # return to the caller
```

if-else

$sp আগের point এখানে দিচ্ছি

n-1 call করা লাগবে তাই arg কে -1 করে দিচ্ছি।

নতুন func. call হয় নাই। ra change হয় নাই তাই ra load করি নাই

$v0 তে return value টা থাকবে fact এর

function call হয়েছে তাই ra load করলাম। আর arg. change করে n-1 করেছি। আগের arg. stack এ ছিল সেটা load করবো।

2 টা জিনিস preserve করা লাগে।

- argument
- return address.

} 4 × 2 → 8 byte storage লাগবে।

slti → set less than immediate

const থাকে তাই

$ $a_0 = n$

$ $t0 = 1$ $n < 1$

$ $t0 = 0$ $n \geq 1$

Pseudo direct addressing

J 25000
(jump)

J-Format

- To support long jump to a remote procedure address
  32 bit

2	10000
6 bits	26 bits

opcode → কোন format

Jumping Address → আরো 6 bit লাগবে
26<<2 [last 2 bit -0 থাকে মেটা বাদ দিয়েই রাখি (right shift)]

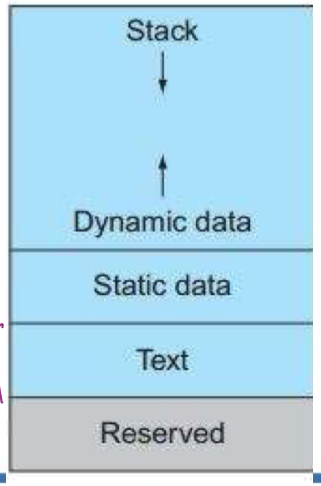
28 bit

বাকি 32 - 28 = 4 bit পাবো PC তে

PC - 31 30 29 28 (27 - 0)

MSB 4 bit

current execution এর address থাকে



এই area র মধ্যেই code গুলো থাকে।

fppt.com

→ 2^{28} 256 MB দূরে একটা program ঝামেলা ছাড়া থাকে না।

32 bit দূরে যদি থাকেও address register এ রেখে jr $r দিবো

lw
sw } 4byte

char → 1 byte এর জন্য cmd লাগবে।

ASCII representation of characters

- ASCII stands for *American Standard Code for Information Interchange*
- Uses 1 byte to represent a character

ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter	ASCII value	Char-acter
32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	DEL

1 Byte Memory Operation

- ASCII demands 1-byte memory operation
- MIPS supports

signed

```
lb   $t0,0($sp)   #Reads 1 byte from memory and stores in the lowest (rightmost) byte of $t0
sb   $t0,0($gp)   #Reads 1 byte from the lowest (rightmost) byte of $t0 and stores in the memory
```

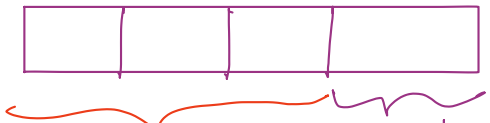
load করে 4byte/32bit register এ write করছি।

Unsigned Version:

Load Byte: lbu

Store Byte: Not available in MIPS

যদি a load করি msb তে 1 বাকি 24bit 1 হয়ে যাবে। so value change হয়ে যাচ্ছে। তখন lbu করবো।

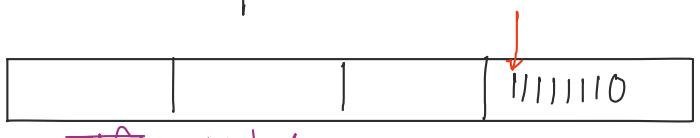


sign extension হবে MSB তে যে bit থাকে সেটা extend করে।

load হবে এখানে

sbu নেই। → store করার সময় signed/unsigned এর কোনো ঝামেলা নেই। সেখানে কোনো operation নেই। just 8bits store.

-2 load করা হলো →



বাকি 24bit এ 1 হবে।

Dealing with Strings

- Three commonly available strategies
  1. ➢ the first position of the string is reserved to give the length of a string
  2. ➢ an accompanying variable has the length of the string
  3. ➢ the last position of a string is indicated by a character used to mark the end of a string – ৫ তে follow করা হয়।

Dealing with String: Example

- ASCII demands 1-byte memory operation
- MIPS supports

```
lb   $t0,0($sp)       #Reads 1 byte from memory and stores in the lowest (rightmost) byte of $t0
sb   $t0,0($gp)       #Reads 1 byte from the lowest (rightmost) byte of $t0 and stores in the memory
```

Unsigned Version:

Load Byte: lbu

Store Byte: Not available in MIPS

Dealing with String: An Example

```
void strcpy (char x[], char y[])
{
    int i;
    i = 0;
    while ((x[i] = y[i]) != '\0')
        i += 1;
}
```

x,y দুটোই array। memory তে আছে। so memory থেকেই read সেখানেই write করা লাগবে।

```
strcpy: addi $sp,$sp,-4          # adjust stack for 1 more item
        sw $s0, 0($sp)           # save $s0
        add $s0,$zero,$zero      # i = 0 + 0
L1:     add $t1,$s0,$a1          # address of y[i] in $t1
        lbu $t2, 0($t1)          # $t2 = y[i]
        add $t3,$s0,$a0          # address of x[i] in $t3
        sb $t2, 0($t3)           # x[i] = y[i]
        beq $t2,$zero,L2         # if y[i] == 0, go to L2
        addi $s0, $s0,1          # i = i + 1
        j L1                     # go to L1. While loop continues
L2:     lw $s0, 0($sp)           # End of string. Restore old $s0
        addi $sp,$sp,4           # pop 1 word off stack
        jr $ra                   # return
```

string এর operation

y এর base address থাকে।

x এর base address

String Copy Example

- MIPS code:

```
strcpy:
    addi $sp, $sp, -4       # adjust stack for 1 item
    sw   $s0, 0($sp)        # save $s0
    add  $s0, $zero, $zero  # i = 0
L1: add  $t1, $s0, $a1      # addr of y[i] in $t1
    lbu  $t2, 0($t1)        # $t2 = y[i]
    add  $t3, $s0, $a0      # addr of x[i] in $t3
    sb   $t2, 0($t3)        # x[i] = y[i]
    beq  $t2, $zero, L2     # exit loop if y[i] == 0
    addi $s0, $s0, 1        # i = i + 1
    j    L1                 # next iteration of loop
L2: lw   $s0, 0($sp)        # restore saved $s0
    addi $sp, $sp, 4        # pop 1 item from stack
    jr   $ra                # and return
```

Dealing with Multiple Languages

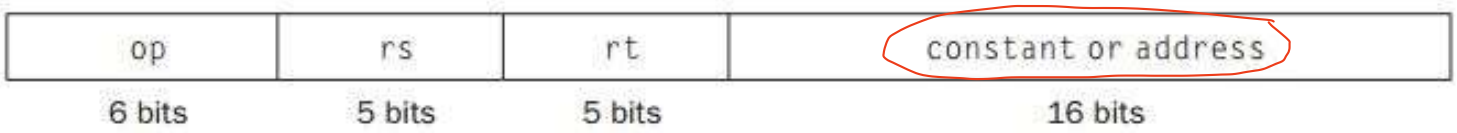
- Unicode, a universal encoding, supports alphabets of most human languages
- Java uses Unicode → 2 byte লাগে
- Requires 16 bits to represent a character
- Operations required to load and store 16bits (halfwords)
  - Load halfword: lh
  - Load halfword unsigned: lhu
  - Store halfword: sh

4 byte word
2 byte halfword

shu → নেই।

Constant Size: Is larger feasible?

- The I-format is used by both immediate and memory data transfer operations



PC+4 ↑ 2^{15} ↓ 2^{15}

- More than 16 bit long constant?
  - We have two options in hand
    1. • Supporting short constant in 1 instruction and deal long constant with additional instructions
    2. • Supporting long constant in 2 instructions
  - We opted for the first option (to exploit the benefit of common case fast)
- Are we limited to use 16 bit constants (-32,768 to 32,767)?
- No ☺ Use two instructions to populate a register with 32 bit constant

branching এ use করি, ($\pm 2^{15}$) instructions পর্যন্ত যাওয়া যাবে।

২য় add এ 16 bits, 2nd add এ 16 bit

Manipulating Larger Constant

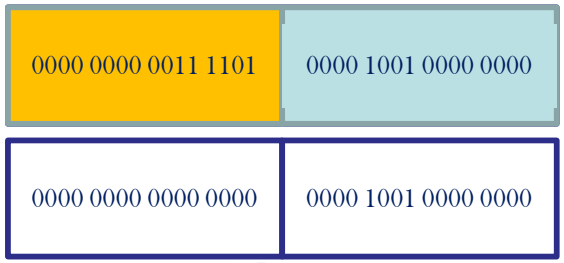
- Populating a register with 32 bit constant
  - Load Upper Immediate (lui) : Loads upper 16 bits
  - Or Immediate (ori): : Loads lower 16 bits
- Example: x=y+4000000
- 4000000 = 0000 0000 0011 1101 0000 1001 0000 0000

61 2304

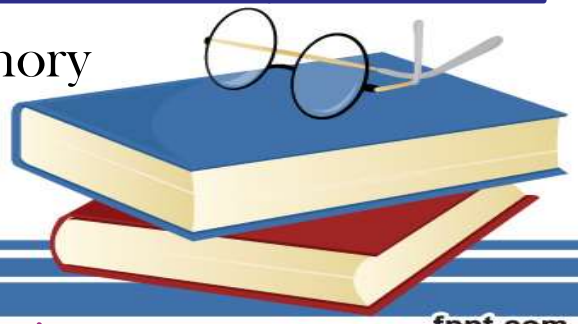
No sign extension for logical operations but it happens for arithmetic operations

MSB load (16bit)

```
lui $s0, 61      upper part load     // 61 = 0000 0000 0011 1101
ori $s0, 2304    পরের 16 bit add     // 2304= 0000 1001 0000 0000
add $s1, $t0, $s0
```



- Think how to use 32 bit address to access data from memory



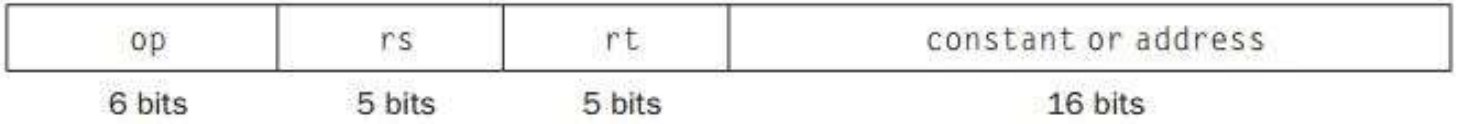
addi $s0, $zero, 2304 → left এ 0 দিয়ে fill
negative value হলে → left এ 1 দিয়ে fill. } sign extension

Addressing in Branches

I format

- PC is 32 bit but the address in conditional branching (if-else, loop etc.) is16 bit

$$bne\ \$s0, \$s1, Exit \qquad \#go\ to\ Exit\ if\ \$s0 \neq s1$$



  - Forces the program instruction count within 2^{16}instructions (2^{18}Bytes=128 KB)
- Most of the conditional branches jump nearby (weather forward or backward)
- Use PC relative addressing to access forward (PC + relative constant address), or backward (PC – relative constant address) location
- Can support branching up to $\pm 2^{15}$relative constant address value
- PC normally increments 1 word (4 bytes) after every instruction
- The addressing should be (PC+4 + relative constant address) for forward access and (PC + 4 – relative constant address) for backward access

Addressing in Jumps

- PC is 32 bit but the address in Jumps is16 bit

- PC is 32 bit but the address in Jumps is16 bit

  - Forces the program instruction count within 2^{26}instructions (2^{28})

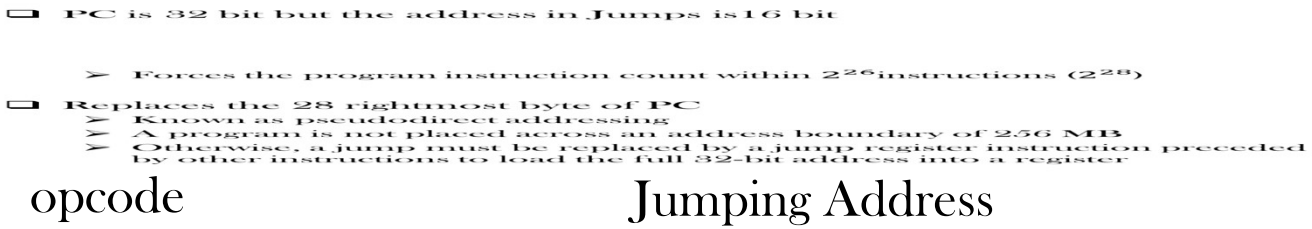
- Replaces the 28 rightmost byte of PC
  - Known as pseudodirect addressing
  - A program is not placed across an address boundary of 256 MB
  - Otherwise, a jump must be replaced by a jump register instruction preceded by other instructions to load the full 32-bit address into a register

opcode Jumping Address

- Forces the program instruction count within 2^{26}instructions (2^{28})

Addressing in Jumps

- PC is 32 bit but the address in Jumps is16 bit



  - Forces the program instruction count within 2^{26}instructions (2^{28})

- Replaces the 28 rightmost byte of PC
  - Known as pseudodirect addressing
  - A program is not placed across an address boundary of 256 MB
  - Otherwise, a jump must be replaced by a jump register instruction preceded by other instructions to load the full 32-bit address into a register

address register ব্যবহার

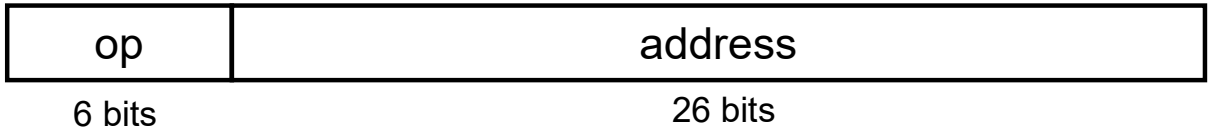
jr ___

Food for Thought

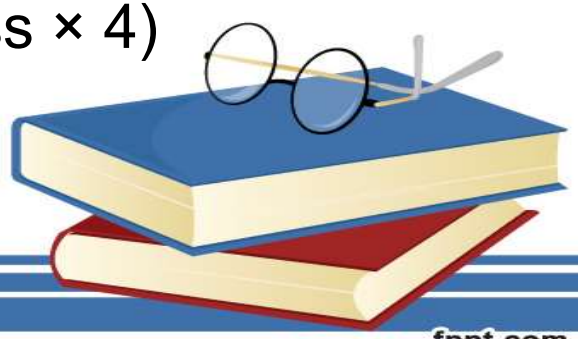
- Can you reverse engineer a binary?
- Is your IP secured?

Jump Addressing

- Jump (`j` and `jal`) targets could be anywhere in text segment
  - Encode full address in instruction



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



xm ৫ দিতে পারে।
code এর corresponding machine code

Target Addressing Example

- Loop code from earlier example
  - Assume Loop at location 80000

$s1, $t1 = 9 দেওয়া থাকলে
∴ $t0 = 8 $s0
$t2 = 10 $s2

```
Loop: sll  $t1, $s3, 2     80000
      add  $t1, $t1, $s6   80004
      lw   $t0, 0($t1)     80008
      bne  $t0, $s5,Exit   80012
      addi $s3, $s3, 1     80016
      j    Loop            80020
Exit: …                    80024
```

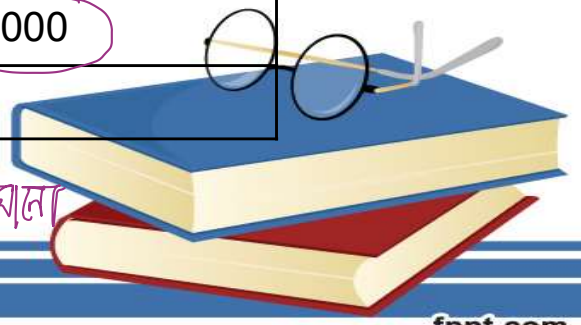
Machine code

0	0	19	9	4	0
0	9	22	9	0	32
35	9	8	0		
5	8	21	2		
8	19	19	1		
2	20000				

PC
PC+4
1
2
ins. count

PC+4 এর respect ৪

4 দ্বারা divide করে বসানো



80012 execute হওয়ার সময়
PC 80016 কে point (by default next ins. execute হতো তাই)

Branching Far Away

- If branch target is too far to encode with 16-bit offset, assembler rewrites the code
- Example

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

↳ beq দিয়ে লিখলাম

assembler যখন দেখবে 16bit এ হচ্ছে না তখন ins. change হয়ে bne করে নিবে।

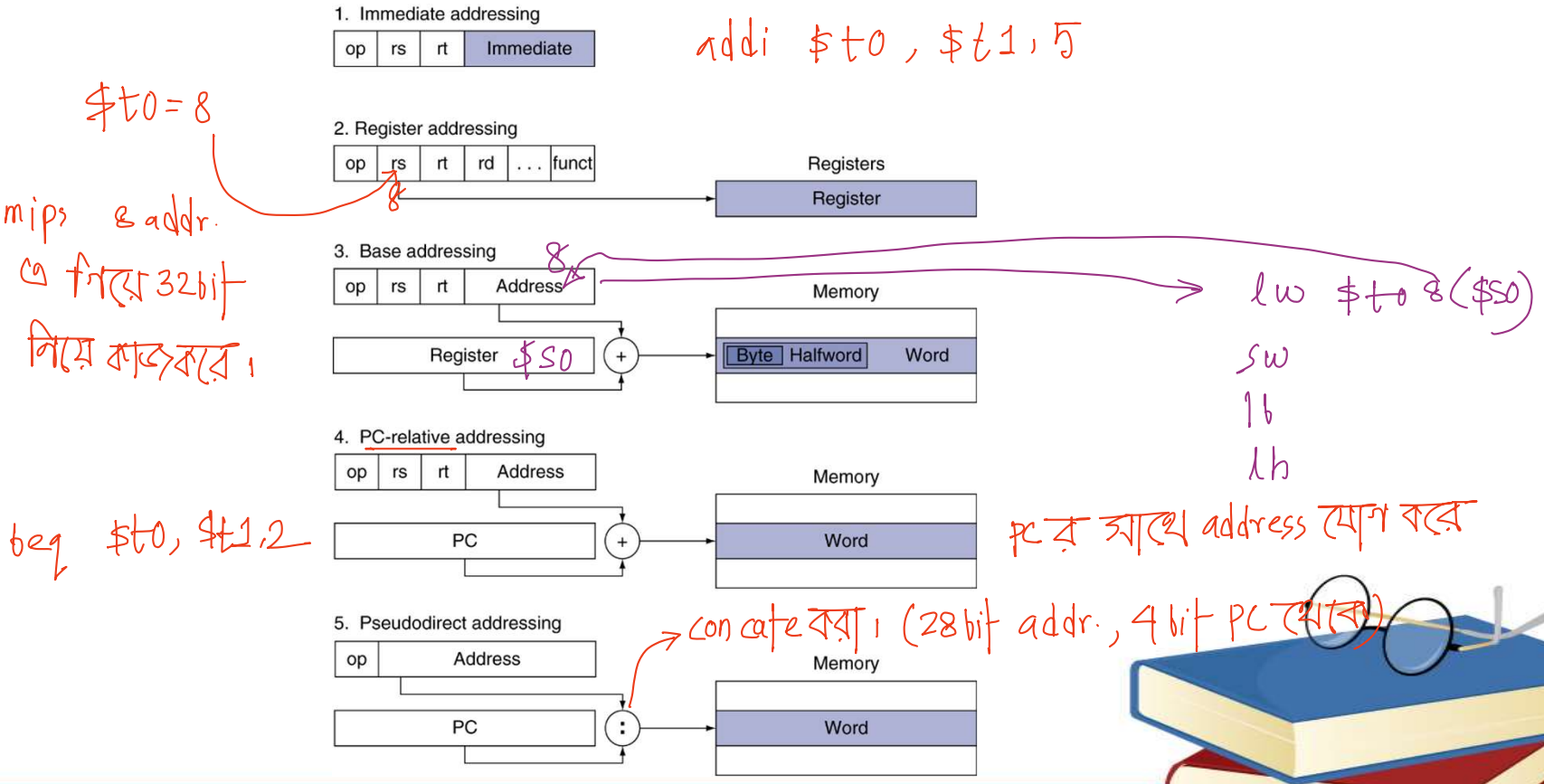
beq দূরে মানে bne কাছে।

26bit পাবো তাহলে addr. রাখার জন্য

আরো বড় addr হলে jump register

explanation
diagram
example

Addressing Mode Summary



Synchronization

- Two processors sharing an area of memory
  - P1 writes, then P2 reads
  - Data race if P1 and P2 don't synchronize
    - Result depends of order of accesses
- Hardware support required
  - Atomic read/write memory operation
  - No other access to the location allowed between the read and write
- Could be a single instruction
  - E.g., atomic swap of register ↔ memory
  - Or an atomic pair of instructions

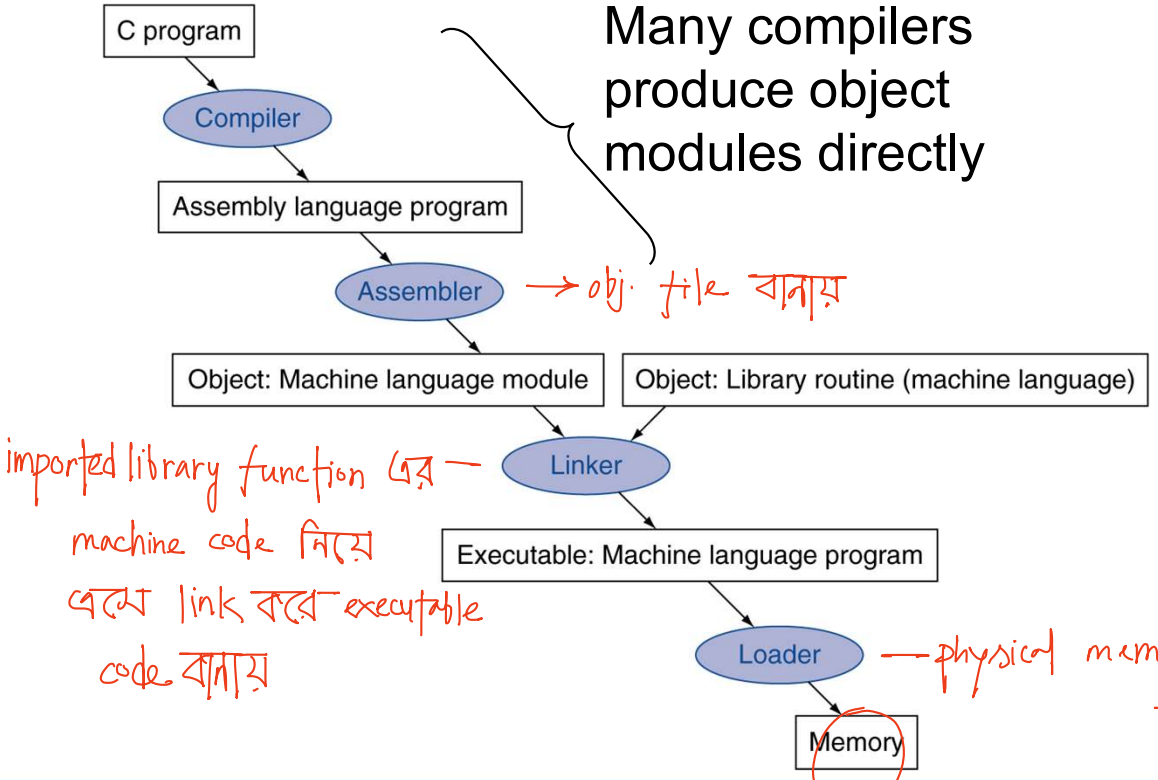
Synchronization in MIPS

- Load linked: `ll rt, offset(rs)`
- Store conditional: `sc rt, offset(rs)`
  - Succeeds if location not changed since the `ll`
    - Returns 1 in rt
  - Fails if location is changed
    - Returns 0 in rt
- Example: atomic swap (to test/set lock variable)

```
try: add $t0,$zero,$s4 ;copy exchange value
     ll  $t1,0($s1)    ;load linked
     sc  $t0,0($s1)    ;store conditional
     beq $t0,$zero,try ;branch store fails
     add $s4,$zero,$t1 ;put load value in $s4
```

OS – (virtual memory থেকে physical memory তে নেয়া) করে

Translation and Startup



→ obj. file বানায়

imported library function এর machine code নিয়ে এসে link করে executable code বানায়

Loader — physical memory যেখানে code execute হবে সেখানে লোড

update পাবোনা

যত file add হবে size বাড়তে থাকবে।

সব bin file add, obj file merge হয়ে exec. create

প্রতিটা program কে abstraction দেওয়া হয় যে তার জন্য full memory আছে।

— Virtual memory

MIPS এ যা লিখি — machine এ convertable

Assembler Pseudoinstructions

- Most assembler instructions represent machine instructions one-to-one
- Pseudoinstructions: figments of the assembler's imagination

MIPS এর নাই এমন ins.

```
move $t0, $t1    →  add $t0, $zero, $t1
blt $t0, $t1, L  →  slt $at, $t0, $t1
                    bne $at, $zero, L
```

Higher level assembly lang এ এমন ins গুলোকে

  - $at (register 1): assembler temporary

lower level (MIPS)

obj file এ machine code + আরো কিছু থাকে।

Producing an Object Module

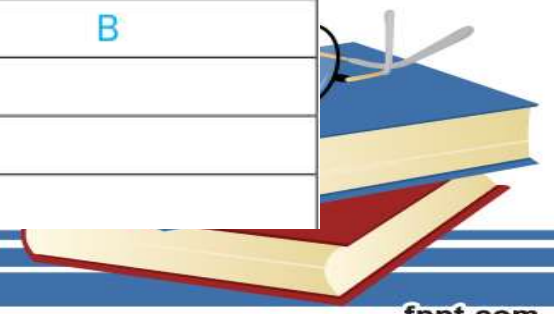
- Assembler (or compiler) translates program into machine instructions
- Provides information for building a complete program from the pieces
  - Header: described contents of object module — procedure name, data size (memory তে এই process কত space নিবে), text seg. size
  - Text segment: translated instructions — machine code থাকে
  - Static data segment: data allocated for the life of the program
  - Relocation info: for contents that depend on absolute location of loaded program — A procedure B কে call করে, B এর addr. এখানে থাকবে।
  - Symbol table: global definitions and external refs — code এ যত symbol থাকে সব নিয়ে Hash table (detailed)
  - Debug info: for associating with source code

Linking Object Modules

- Produces an executable image
  1. Merges segments
  2. Resolve labels (determine their addresses)
  3. Patch location-dependent and external refs
- Could leave location dependencies for fixing by a relocating loader
  - But with virtual memory, no need to do this
  - Program can be loaded into absolute location in virtual memory space

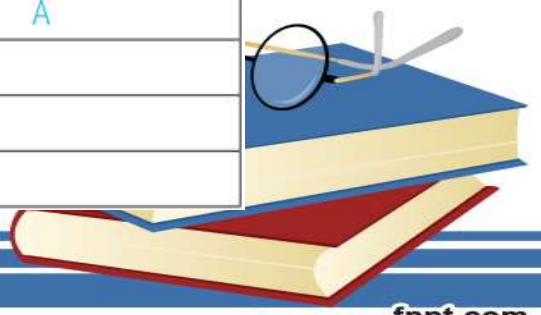
Object 1

Object file header			
	Name	Procedure A	
	Text size	100_{hex}	
	Data size	20_{hex}	
Text segment	Address	Instruction	
	0	lw $a0, 0($gp)	
	4	jal 0	
	...	...	
Data segment	0	(X)	
	...	...	
Relocation information	Address	Instruction type	Dependency
	0	lw	X
	4	jal	B
Symbol table	Label	Address	
	X	–	
	B	–	

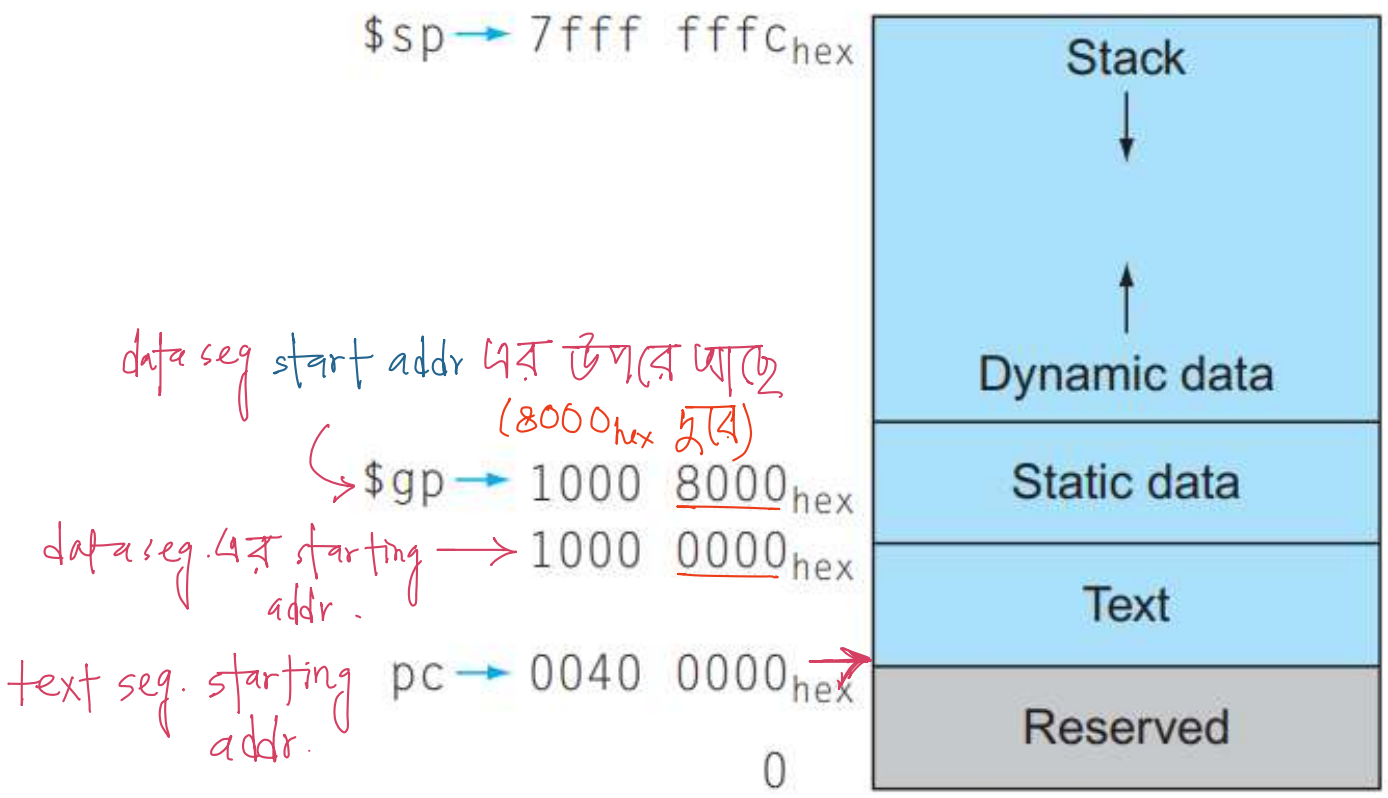


Object 2

Object file header			
	Name	Procedure B	
	Text size	200_{hex}	
	Data size	30_{hex}	
Text segment	Address	Instruction	
	0	sw $a1, 0($gp)	
	4	jal 0 A	
	...	...	
Data segment	0	(Y)	
	...	...	
Relocation information	Address	Instruction type	Dependency
	0	sw	Y
	4	jal	A
Symbol table	Label	Address	
	Y	–	
	A	–	

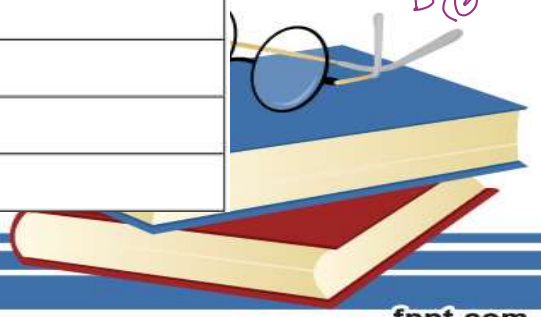


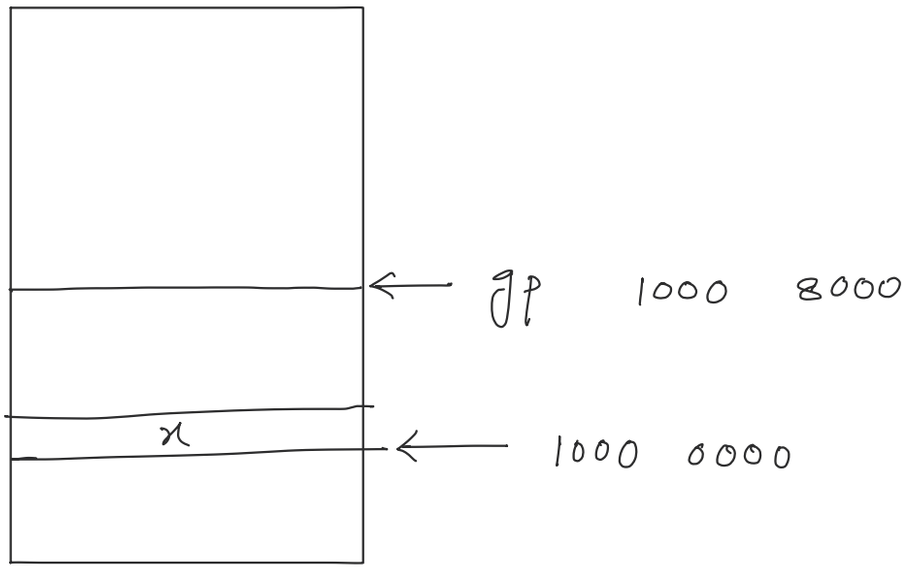
সবার জন্য virtual memory এটা



Merged Object

Executable file header		
	Text size	300_{hex}
	Data size	50_{hex}
Text segment	Address	Instruction
	$0040\ 0000_{hex}$	lw $a0, 8000_{hex}($gp)
A	$0040\ 0004_{hex}$	jal $40\ 0100_{hex}$
	...	...
	$0040\ 0100_{hex}$	sw $a1, 8020_{hex}($gp)
(300 hex) B	$0040\ 0104_{hex}$	jal $40\ 0000_{hex}$
	...	...
Data segment	Address	
	$1000\ 0000_{hex}$	(X)
	...	...
	$1000\ 0020_{hex}$	(Y)
	...	...





gp → higher address এ আছে।

x নিচে।

− 8000 hex

↳ 16's complement নিতে হবে।

= 8000_{hex}

8000
−20
7980

−7980 → 16's complement 8020

exam: obj file দেওয়া থাকবে merge করা লাগবে।

Loading a Program

- Load from image file on disk into memory
  1. Read header to determine segment sizes
  2. Create virtual address space
  3. Copy text and initialized data into memory
     - Or set page table entries so they can be faulted in
  4. Set up arguments on stack
  5. Initialize registers (including $sp, $fp, $gp)
  6. Jump to startup routine
     - Copies arguments to $a0, ... and calls main
     - When main returns, do exit syscall

library file
↳ সব copy করে না এনে address রাখো so update হলে সেটা পাবো।

Dynamic Linking

- Only link/load library procedure when it is called
  - Requires procedure code to be relocatable
  - Avoids image bloat caused by static linking of all (transitively) referenced libraries
  - Automatically picks up new library versions

xm: static, dynamic linking difference

অসুবিধা → updated version এ procedure বাদ দিলে or name change করা হলে
– DLL not found.

Dynamically Linked Libraries (DLL)

- Library routines that are linked to a program during execution
  - To incorporate the updated version of library routines
- In lazy procedure linkage, each routine is linked only when it is called
  - Provides a level of indirection → যেই procedure call করবো সেটার addr. শুধু copy করে। প্রতিবার call এ copy করে না।
  - Nonlocal routine calls a set of dummy entries at the end of the program, with one entry per nonlocal routine. These dummy entries each contain an indirect jump
  - The Dynamic Linker/Loader finds the desired routine, remaps it, and changes the address in the indirect jump location to point to that routine
  - From then, the call to the library routine jump indirectly to routine without the extra hops

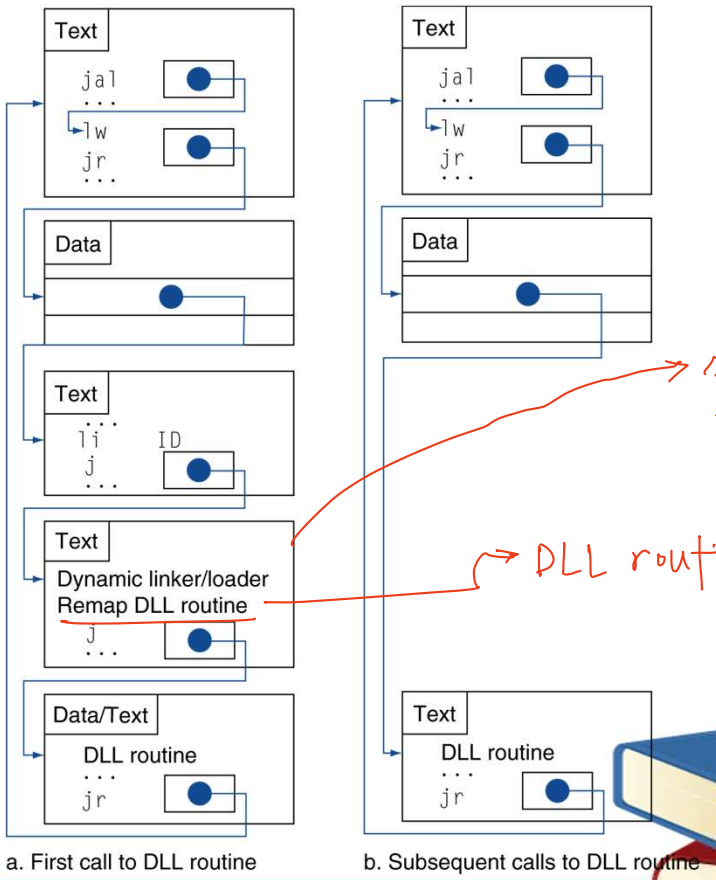
Lazy Linkage

Indirection table

Stub: Loads routine ID,
Jump to linker/loader

Linker/loader code

Dynamically
mapped code

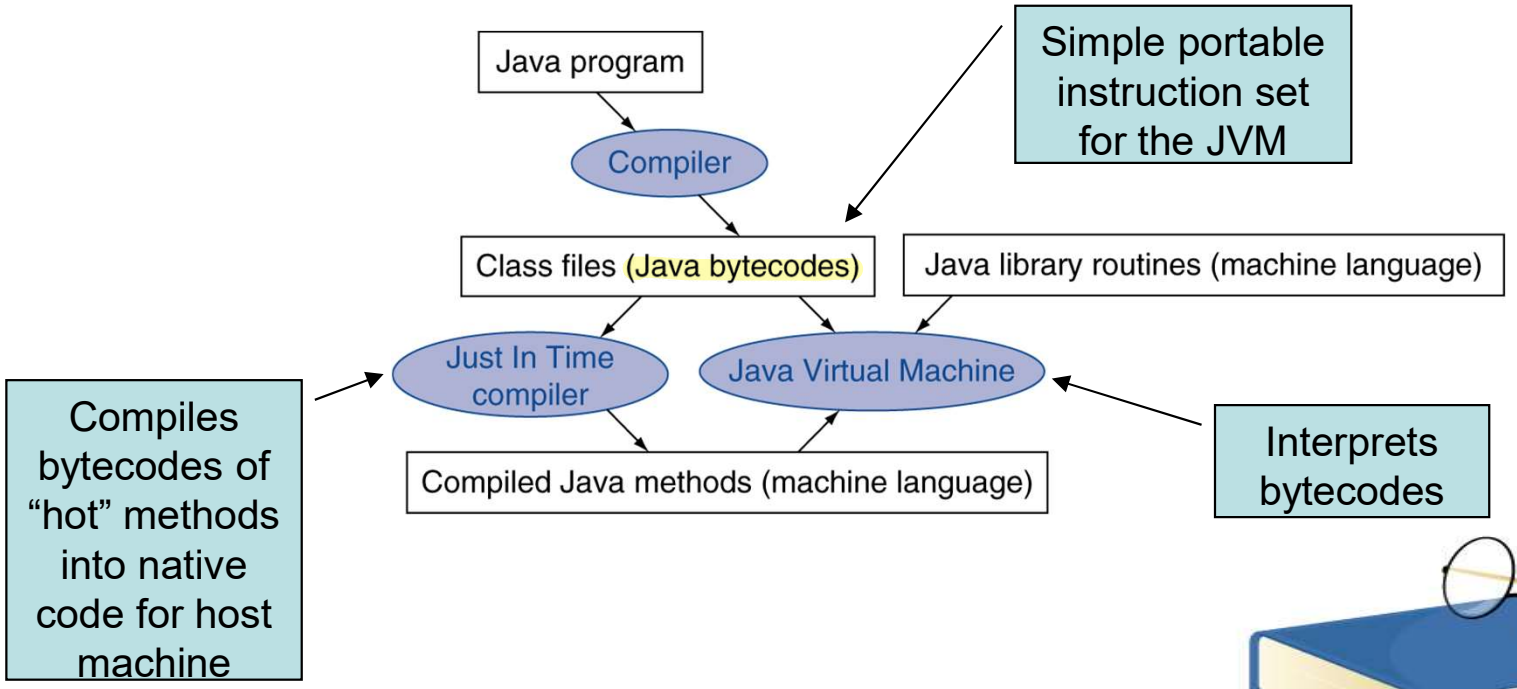


Starting a Java Program

- Java programs ensure portability sacrificing some performance
- Compiled first in to an easy-to-interpret instruction set: Java bytecode
- A software interpreter, called Java Virtual Machine (JVM) can execute Java byte code
  - This process is slow
- Just In Time Compiler (JIT) makes it faster
  - Statistically identify the commonly used (hot) methods
  - Compiles these methods into native instruction set

commonly used method গুলো → machine code এ convert করে রেখে দেয়
বাকিগুলো interpret করে
↓
make the common case faster

Starting Java Applications



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

The Procedure Swap

```
swap: sll $t1, $a1, 2    # $t1 = k * 4
      add $t1, $a0, $t1  # $t1 = v+(k*4)
                         #   (address of v[k])
      lw $t0, 0($t1)     # $t0 (temp) = v[k]
      lw $t2, 4($t1)     # $t2 = v[k+1]
      sw $t2, 0($t1)     # v[k] = $t2 (v[k+1])
      sw $t0, 4($t1)     # v[k+1] = $t0 (temp)
      jr $ra             # return to calling routine
```

The Sort Procedure in C

- Non-leaf (calls swap)

```
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);
    }
  }
}
```

  - v in $a0, k in $a1, i in $s0, j in $s1

The Procedure Body

```
        move $s2, $a0           # save $a0 into $s2
        move $s3, $a1           # save $a1 into $s3
        move $s0, $zero         # i = 0
for1tst: slt  $t0, $s0, $s3     # $t0 = 0 if $s0 ≥ $s3 (i ≥ n)
        beq  $t0, $zero, exit1  # go to exit1 if $s0 ≥ $s3 (i ≥ n)
        addi $s1, $s0, -1       # j = i - 1
for2tst: slti $t0, $s1, 0       # $t0 = 1 if $s1 < 0 (j < 0)
        bne  $t0, $zero, exit2  # go to exit2 if $s1 < 0 (j < 0)
        sll  $t1, $s1, 2        # $t1 = j * 4
        add  $t2, $s2, $t1      # $t2 = v + (j * 4)
        lw   $t3, 0($t2)        # $t3 = v[j]
        lw   $t4, 4($t2)        # $t4 = v[j + 1]
        slt  $t0, $t4, $t3      # $t0 = 0 if $t4 ≥ $t3
        beq  $t0, $zero, exit2  # go to exit2 if $t4 ≥ $t3
        move $a0, $s2           # 1st param of swap is v (old $a0)
        move $a1, $s1           # 2nd param of swap is j
        jal  swap               # call swap procedure
        addi $s1, $s1, -1       # j -= 1
        j    for2tst            # jump to test of inner loop
exit2:  addi $s0, $s0, 1        # i += 1
        j    for1tst            # jump to test of outer loop
```

Move params

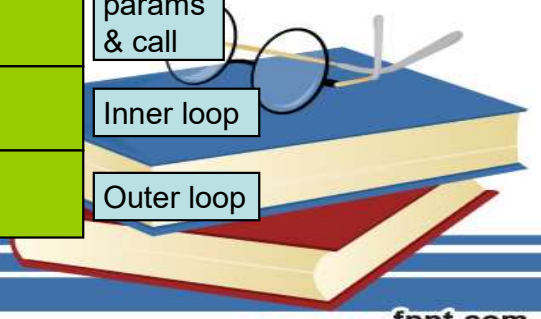
Outer loop

Inner loop

Pass params & call

Inner loop

Outer loop

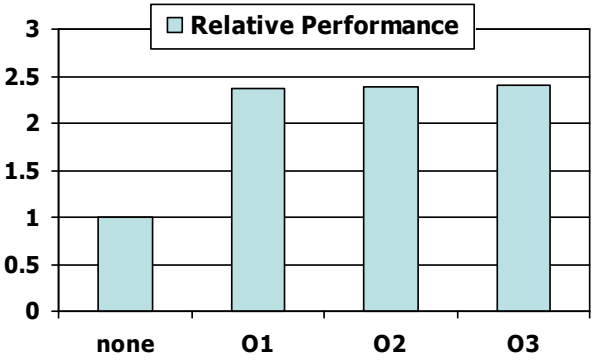


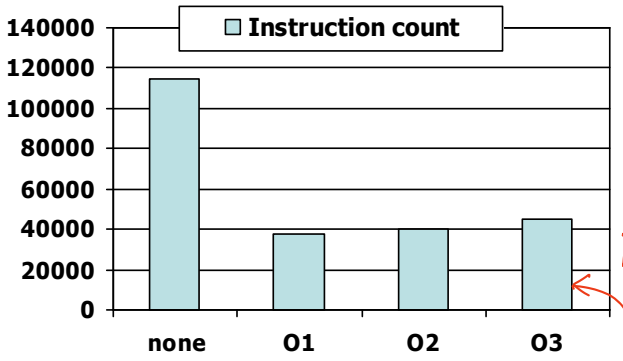
The Full Procedure

```
sort:   addi $sp,$sp, -20      # make room on stack for 5 registers
        sw $ra, 16($sp)        # save $ra on stack
        sw $s3,12($sp)         # save $s3 on stack
        sw $s2, 8($sp)         # save $s2 on stack
        sw $s1, 4($sp)         # save $s1 on stack
        sw $s0, 0($sp)         # save $s0 on stack
        …                      # procedure body
        …
        exit1: lw $s0, 0($sp)  # restore $s0 from stack
        lw $s1, 4($sp)         # restore $s1 from stack
        lw $s2, 8($sp)         # restore $s2 from stack
        lw $s3,12($sp)         # restore $s3 from stack
        lw $ra,16($sp)         # restore $ra from stack
        addi $sp,$sp, 20       # restore stack pointer
        jr $ra                 # return to calling routine
```

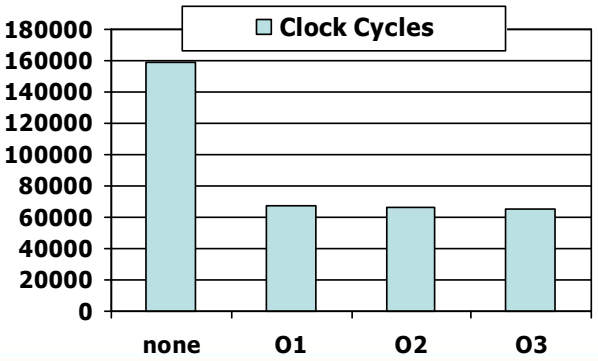
Effect of Compiler Optimization

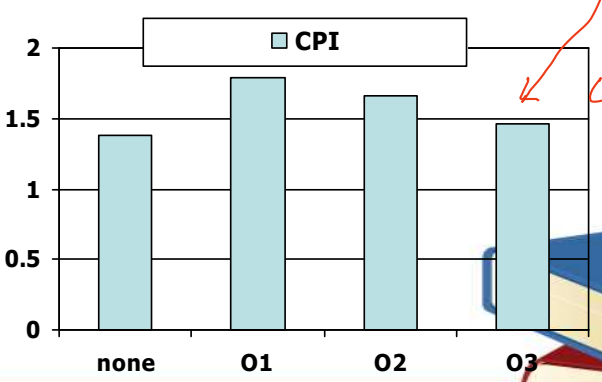
Compiled with gcc for Pentium 4 under Linux





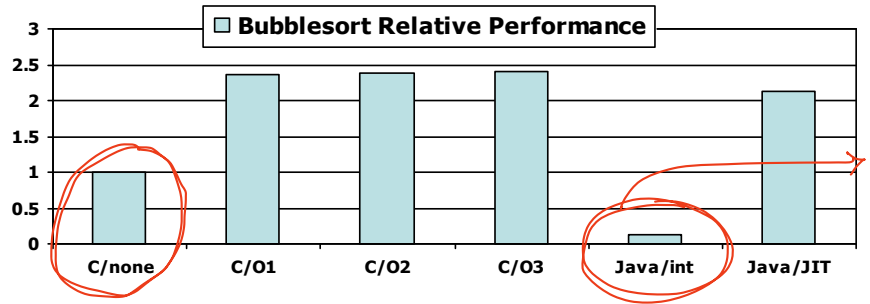
ins. count বাড়লেও

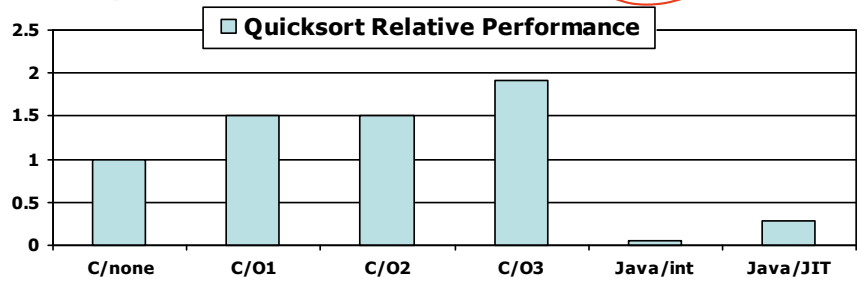


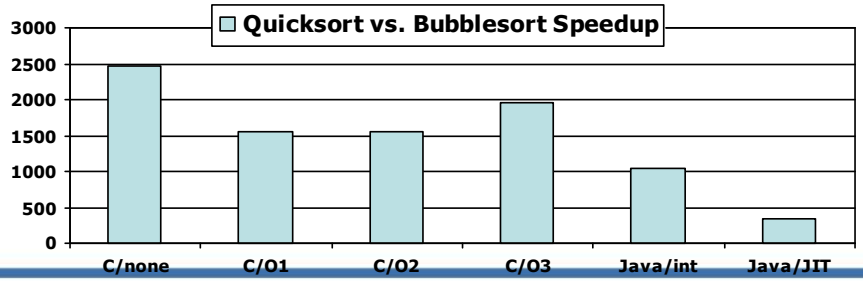


CPI কম

Effect of Language and Algorithm







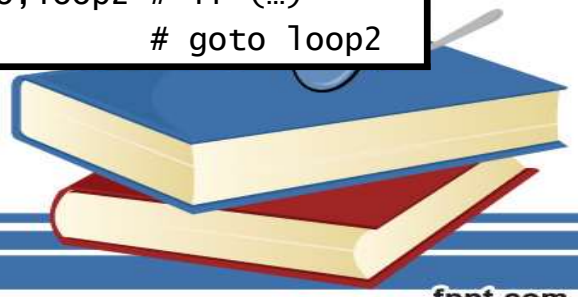
Example: Clearing an Array

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

```
        move $t0,$zero   # i = 0
loop1:  sll $t1,$t0,2    # $t1 = i * 4
        add $t2,$a0,$t1  # $t2 =
                         #   &array[i]
        sw $zero, 0($t2) # array[i] = 0
        addi $t0,$t0,1   # i = i + 1
        slt $t3,$t0,$a1  # $t3 =
                         #   (i < size)
        bne $t3,$zero,loop1 # if (…)
                            # 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 = array의 last addr.
                         #   &array[size]
loop2:  sw $zero,0($t0)  # Memory[p] = 0
        addi $t0,$t0,4   # p = p + 4
        slt $t3,$t0,$t2  # $t3 =
                         #(p<&array[size])
        bne $t3,$zero,loop2 # if (…)
                            # goto loop2
```



Comparison of Array vs. Ptr

- Multiply "strength reduced" to shift
- Array version requires shift to be inside loop
  - Part of index calculation for incremented i
  - c.f. incrementing pointer
- Compiler can achieve same effect as manual use of pointers
  - Induction variable elimination
  - Better to make program clearer and safer

MIPS (RISC) Design Principles In Summary

- **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
  - limited number of addressing modes
- **Good design demands good compromises**
  - Three instruction formats
- **Make the common case fast**
  - arithmetic operands using the registers
  - Saving the commonly used registers into stack
  - PC-relative addressing
  - allow instructions to contain immediate operands

tf: কোন decision নিলে কোন design pattern follow করছে।

Lessons Learnt

- Instruction count and CPI are not good performance indicators in isolation
- Compiler optimizations are sensitive to the algorithm
- Java/JIT compiled code is significantly faster than JVM interpreted
  - Comparable to optimized C in some cases
- Nothing can fix a dumb algorithm!

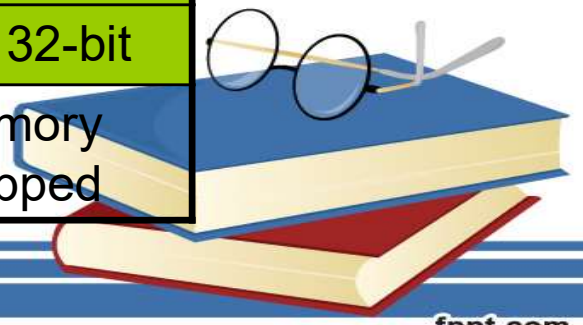
Alternative Architectures

- Design alternative
  - provide more powerful operations
  - goal is to reduce number of instructions executed
  - danger is a slower cycle time and/or a higher CPI

ARM & MIPS Similarities

- ARM: the most popular embedded core
- Similar basic set of instructions to MIPS

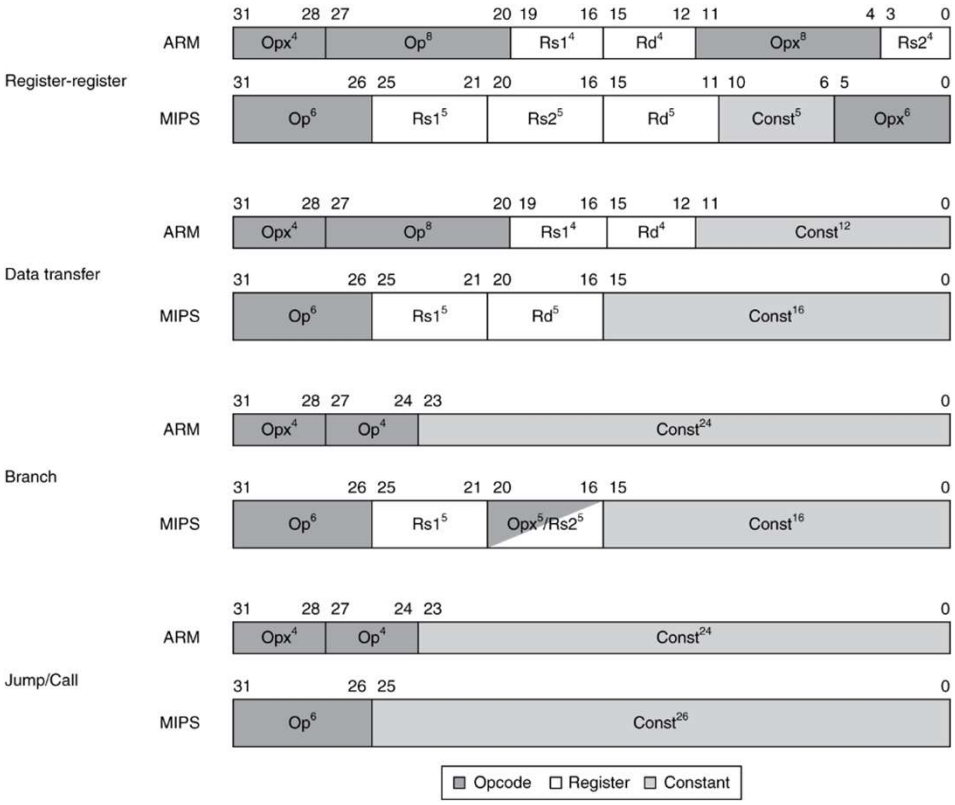
	ARM	MIPS
Date announced	1985	1985
Instruction size	32 bits	32 bits
Address space	32-bit flat	32-bit flat
Data alignment	Aligned	Aligned
Data addressing modes	9	3
Registers	15 × 32-bit	31 × 32-bit
Input/output	Memory mapped	Memory mapped



Compare and Branch in ARM

- Uses condition codes for result of an arithmetic/logical instruction
  - Negative, zero, carry, overflow
  - Compare instructions to set condition codes without keeping the result
- Each instruction can be conditional
  - Top 4 bits of instruction word: condition value
  - Can avoid branches over single instructions

Instruction Encoding



The Intel x86 ISA

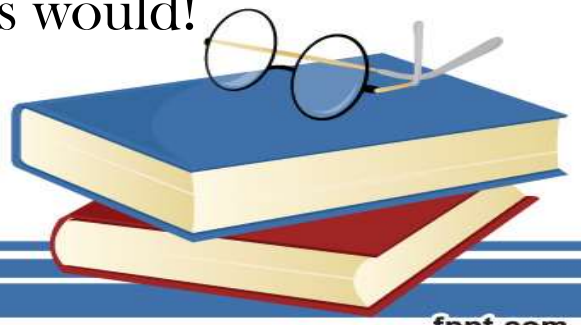
- Evolution with backward compatibility
  - 8080 (1974): 8-bit microprocessor
    - Accumulator, plus 3 index-register pairs
  - 8086 (1978): 16-bit extension to 8080
    - Complex instruction set (CISC)
  - 8087 (1980): floating-point coprocessor
    - Adds FP instructions and register stack
  - 80286 (1982): 24-bit addresses, MMU
    - Segmented memory mapping and protection
  - 80386 (1985): 32-bit extension (now IA-32)
    - Additional addressing modes and operations
    - Paged memory mapping as well as segments

The Intel x86 ISA

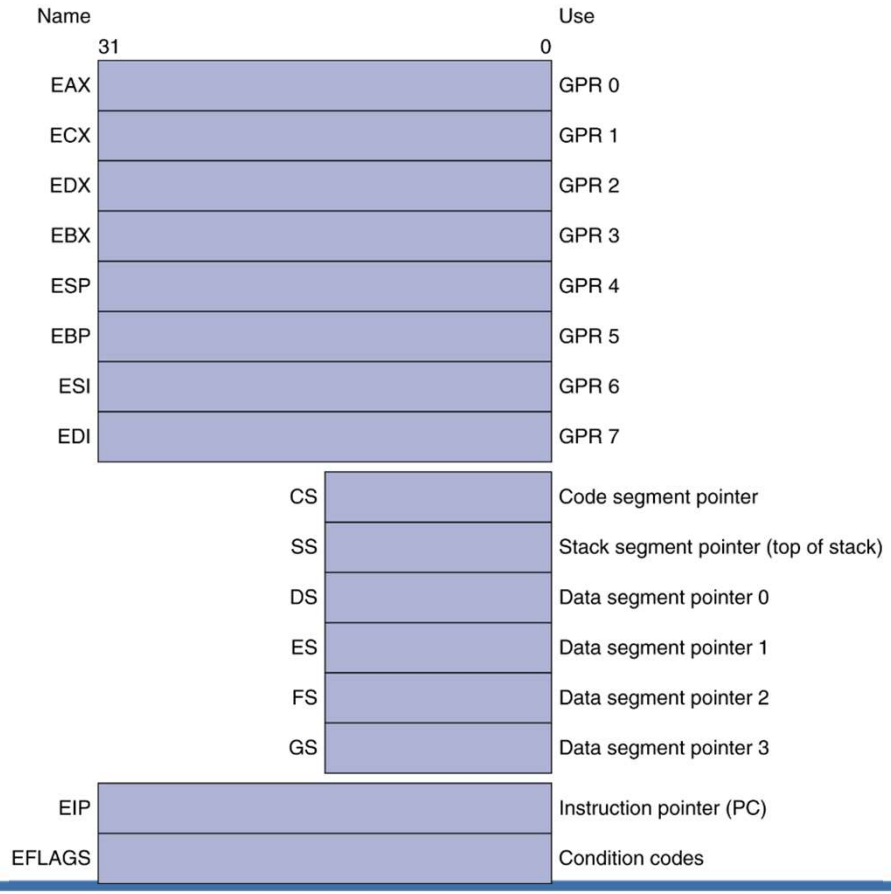
- Further evolution...
  - i486 (1989): pipelined, on-chip caches and FPU
    - Compatible competitors: AMD, Cyrix, ...
  - Pentium (1993): superscalar, 64-bit datapath
    - Later versions added MMX (Multi-Media eXtension) instructions
    - The infamous FDIV bug
  - Pentium Pro (1995), Pentium II (1997)
    - New microarchitecture (see Colwell, *The Pentium Chronicles*)
  - Pentium III (1999)
    - Added SSE (Streaming SIMD Extensions) and associated registers
  - Pentium 4 (2001)
    - New microarchitecture
    - Added SSE2 instructions

The Intel x86 ISA

- And further…
  - AMD64 (2003): extended architecture to 64 bits
  - EM64T – Extended Memory 64 Technology (2004)
    - AMD64 adopted by Intel (with refinements)
    - Added SSE3 instructions
  - Intel Core (2006)
    - Added SSE4 instructions, virtual machine support
  - AMD64 (announced 2007): SSE5 instructions
    - Intel declined to follow, instead…
  - Advanced Vector Extension (announced 2008)
    - Longer SSE registers, more instructions
- If Intel didn't extend with compatibility, its competitors would!
  - Technical elegance ≠ market success



Basic x86 Registers



Basic x86 Addressing Modes

- Two operands per instruction

Source/dest operand	Second source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

- Memory addressing modes
  - Address in register
  - Address = R_{base} + displacement
  - Address = $R_{base} + 2^{scale} \times R_{index}$ (scale = 0, 1, 2, or 3)
  - Address = $R_{base} + 2^{scale} \times R_{index}$ + displacement

x86 Instruction Encoding

a. JE EIP + displacement

4	4	8
JE	Condi-tion	Displacement

b. CALL

8	32
CALL	Offset

c. MOV EBX, [EDI + 45]

6	1	1	8	8
MOV	d	w	r/m Postbyte	Displacement

d. PUSH ESI

5	3
PUSH	Reg

e. ADD EAX, #6765

4	3	1	32
ADD	Reg	w	Immediate

f. TEST EDX, #42

7	1	8	32
TEST	w	Postbyte	Immediate

- Variable length encoding
  - Postfix bytes specify addressing mode
  - Prefix bytes modify operation
    - Operand length, repetition, locking, ...

Implementing IA-32 (Seg:Off)

- Complex instruction set makes implementation difficult
  - Hardware translates instructions to simpler microoperations
    - Simple instructions: 1–1
    - Complex instructions: 1–many
  - Microengine similar to RISC
  - Market share makes this economically viable
- Comparable performance to RISC
  - Compilers avoid complex instructions

ARM v8 Instructions

- In moving to 64-bit, ARM did a complete overhaul
- ARM v8 resembles MIPS
  - Changes from v7:
    - No conditional execution field
    - Immediate field is 12-bit constant
    - Dropped load/store multiple
    - PC is no longer a GPR
    - GPR set expanded to 32
    - Addressing modes work for all word sizes
    - Divide instruction
    - Branch if equal/branch if not equal instructions

Fallacies

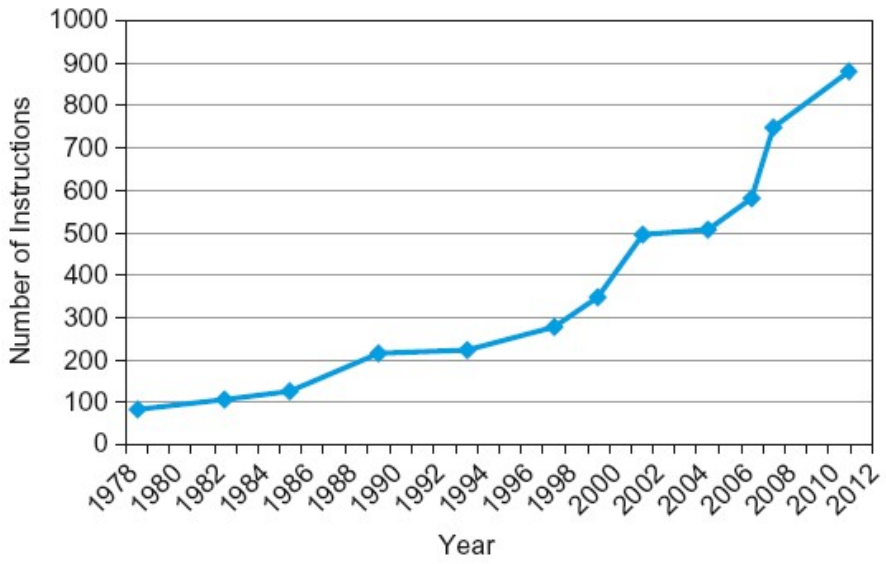
ins. কম but CPI বেশি

- Powerful instruction ⇒ higher performance
  - Fewer instructions required
  - But complex instructions are hard to implement
    - May slow down all instructions, including simple ones
  - Compilers are good at making fast code from simple instructions
- Use assembly code for high performance
  - But modern compilers are better at dealing with modern processors
  - More lines of code ⇒ more errors and less productivity

Fallacies

- Backward compatibility ⇒ instruction set doesn't change
  - But they do accrete more instructions

আগের ins. গুলো থাকবেই
আরো নতুন ins. add হচ্ছে



x86 instruction set

Pitfalls

- Sequential words are not at sequential addresses
  - Increment by 4, not by 1!
- Keeping a pointer to an automatic variable after procedure returns
  - e.g., passing pointer back via an argument
  - Pointer becomes invalid when stack popped

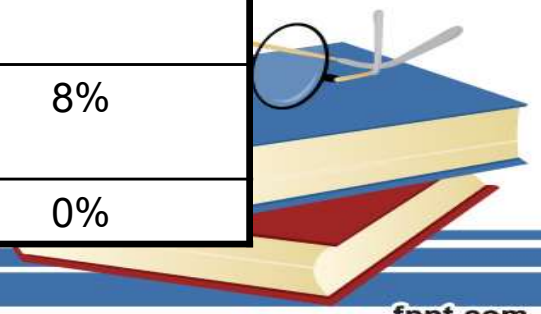
Concluding Remarks

- 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
  - Compiler, assembler, hardware
- MIPS: typical of RISC ISAs
  - c.f. x86

Concluding Remarks

- Measure MIPS instruction executions in benchmark programs
  - Consider making the common case fast
  - Consider compromises

Instruction class	MIPS examples	SPEC2006 Int	SPEC2006 FP
Arithmetic	`add, sub, addi`	16%	48%
Data transfer	`lw, sw, lb, lbu, lh, lhu, sb, lui`	35%	36%
Logical	`and, or, nor, andi, ori, sll, srl`	12%	4%
Cond. Branch	`beq, bne, slt, slti, sltiu`	34%	8%
Jump	`j, jr, jal`	2%	0%



Acknowledgements

- These slides contain material developed and copyright by:
  - Lecture slides by Dr. Tanzima Hashem, Professor, CSE, BUET
  - Lecture slides by Mehnaz Tabassum Mahin, Assistant Professor, CSE, BUET

Thank You ☺