

# Instruction Set Architecture



Instructor: Kalyan Basu

[Basu@cse.uta.edu](mailto:Basu@cse.uta.edu)

# Language

**HLL** : High Level Language Program written by Programming language like C, C++, Java.

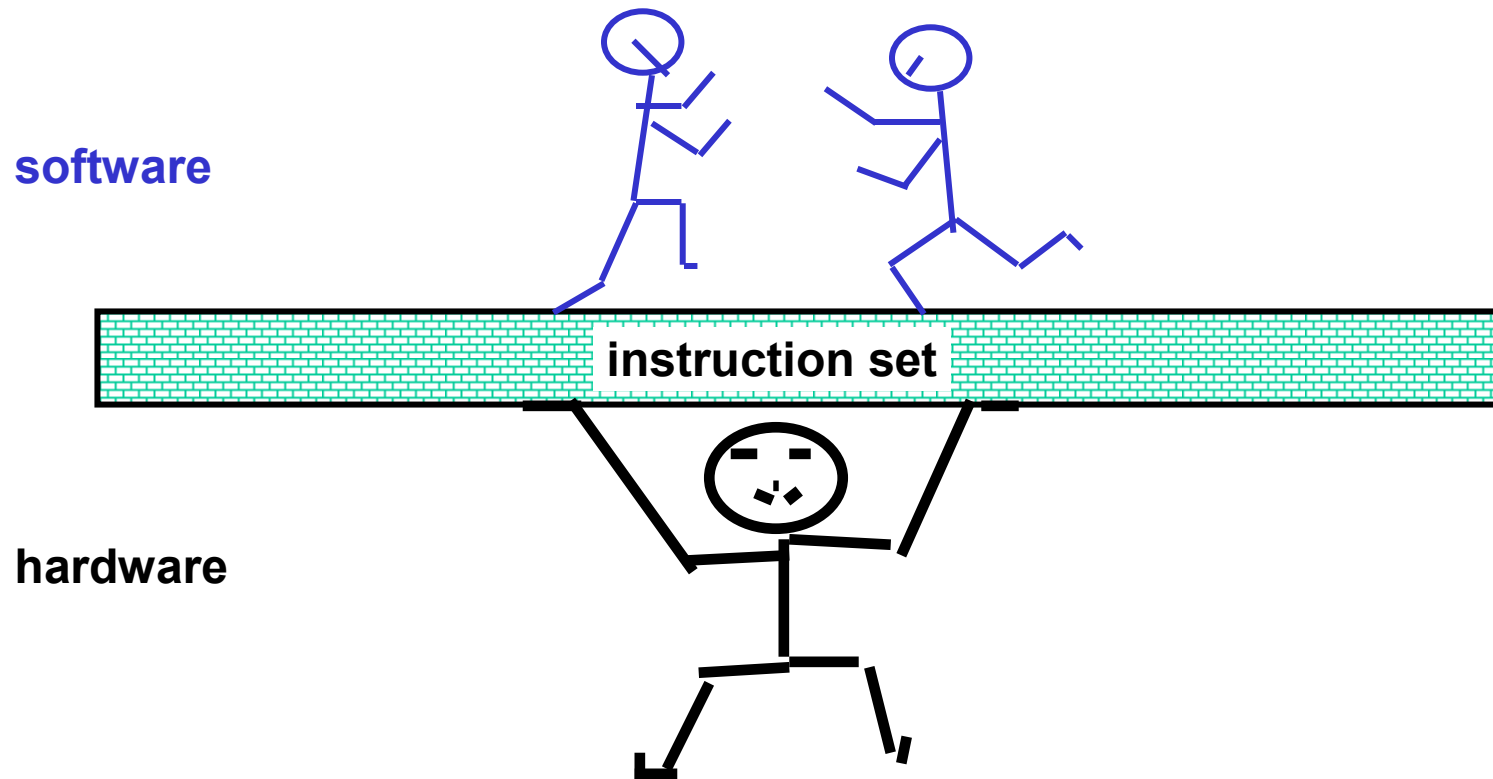
*Sentence*       $a = b + c;$   
 $d = a - e;$

**Assembly Language**: The Pneumonic translation of Binary code into English Language. One to one correspondence between Binary Language and Assembly language. MIPS is the Assembly language used in the book.

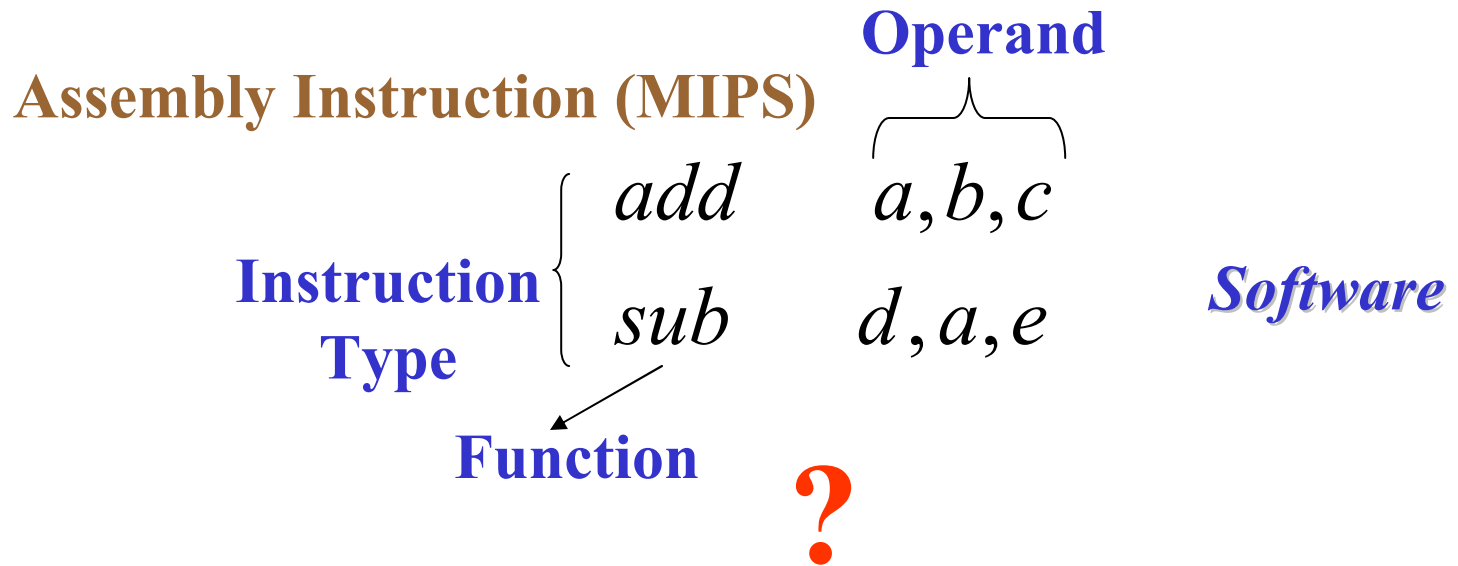
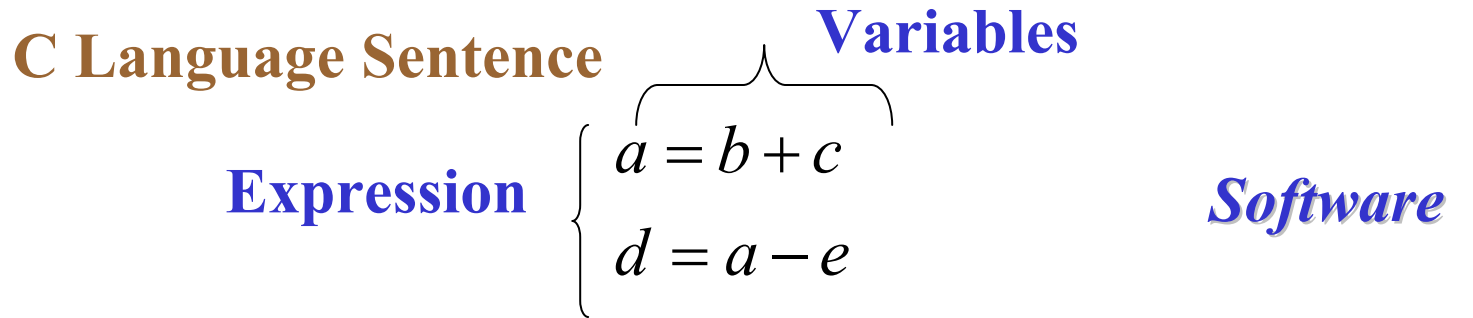
*Instruction*       $add\ a,\ b,\ c$   
 $sub\ d,\ a,\ e$

**Binary Code**: Language expressed by Binary numbers understood by the computer Hardware. Each

# Instruction Set



# Computer Operation



1. Where is a, b, c, d and e
2. Who is doing add and sub

# Hardware Software Interface

*Software*

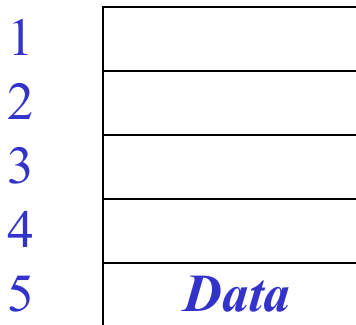


*Software*



1. Where is a, b, c, d and e
2. Who is doing add and sub

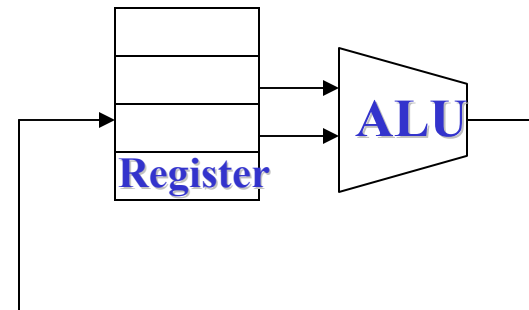
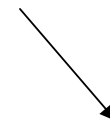
**Memory**



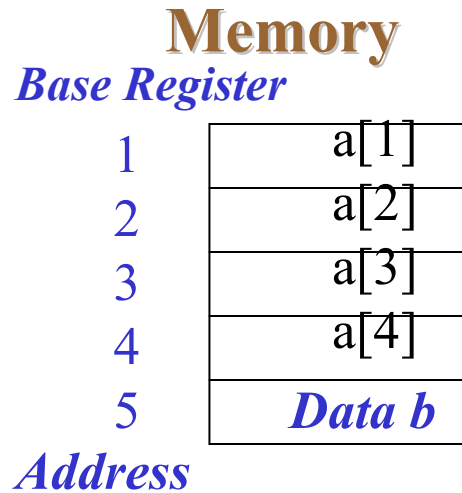
*Address*

*Hardware*

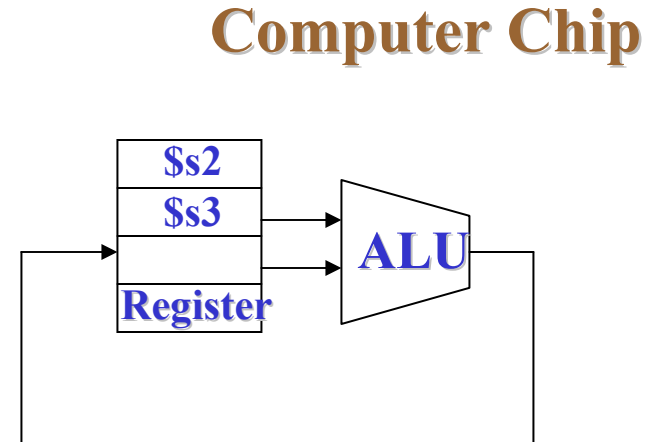
**Computer Chip**



# Hardware Software Interface



*Hardware*



## 1. Where is Data

In Memory (a[1-4], b) Starting address at **Base Register** for an array

In Registers (\$s2, \$s3,...,\$t0) Explicit address

## 2. Who Transfers Data between Memory and Registers

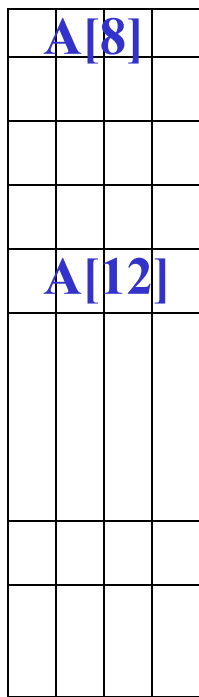
## 3. Who is the Controller of ALU

# HLL to Hardware

## C Language statement

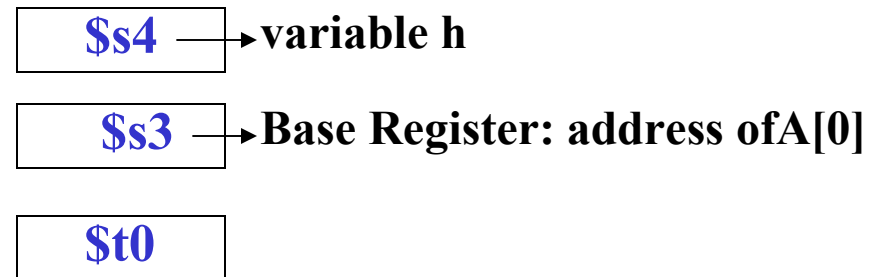
$$A[12] = h + A[8]$$

A: Array  
h: variable



$$A[8] = \$s3 + 32$$

$$A[12] = \$s3 + 48$$



## MIPS Instruction

```
lw    $s2, 32($s3)
add   $t0, $s2, $s4
sw    $t0, 48($s3)
```

4 Bytes per data (32 bits)

# Machine Representation

*lw*     $\$t_0, 32(\$s_3)$

*add*     $\$t_0, \$s_2, \$t_0$

*sw*     $\$t_0, 48(\$s_3)$

Machine is in Binary, but this expressions are not in binary  
We need Binary Translation. **Machine Language**

<i>Op Code (6)</i>	<i>rs (5)</i>	<i>rt (5)</i>	<i>rd (5)</i>	<i>shamt(5)</i>	<i>function(6)</i>
--------------------	---------------	---------------	---------------	-----------------	--------------------

**Op Code**: Operation of the instruction

**rs**: The first source operand register

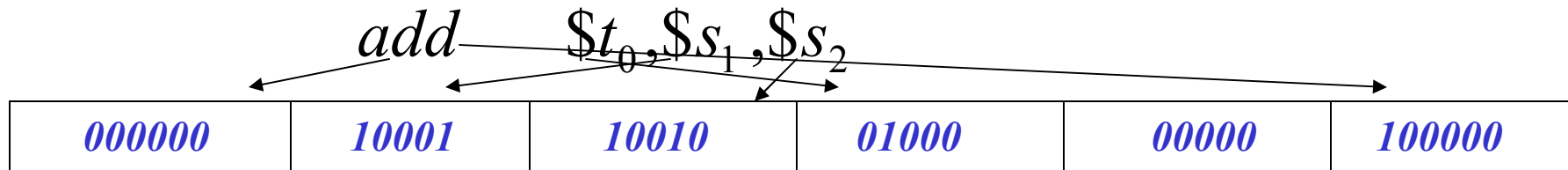
**rt**: The second source operand register

**rd**: Destination operand register. Gets results of the operation

**shamt**: Shift amount for shift instruction

**function**: Selects the functions within a opcode field.

# Hexadecimal



It is difficult to remember this large binary sequence.

Hexadecimal conversion is used. It is a number system to the base 16.

So 4 bits represent one symbol and there are 16 symbols.

$$0_{hex} = 0000_2$$

$$1_{hex} = 0001_2$$

$$9_{hex} = 1001_2$$

$$a_{hex} = 1010_2$$

\*

$$f_{hex} = 1111_2$$

0000 0010 0011 0010 0100 0000 0010 0000  
 0 2 3 2 8 0 2 0

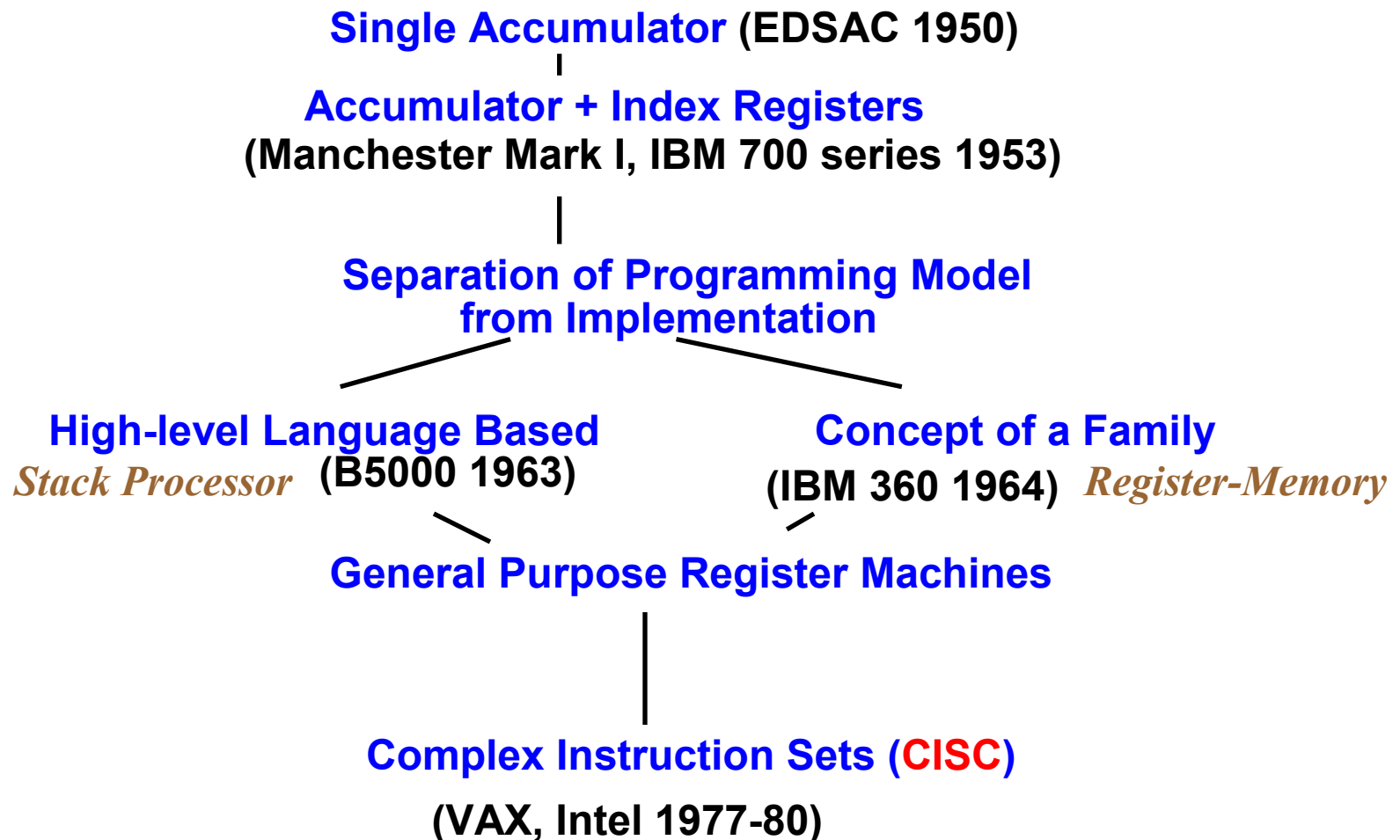


02328020<sub>hex</sub>

# Instruction Set Architecture

- A computer architect must decide on the set of instructions that are **executable in hardware** on their designed machine.
- These instructions must:
  - satisfy the **design goals** of the target machine in terms of cost and performance, and
  - support all the **language constructs** specified in a high-level or assembly-level language to be run on the target machine.

# Evolution of Instruction Sets



# Architecture Thrust

Some Important Questions:

1. What is the *nature of the programs*
  1. Simplicity **Tanenbaum**
  2. Structured
  3. Integer or floating points
2. High Level *Language Mapping* (CISC)
  1. Simplify compilation by easy mapping to instruction
  2. How to reduce code size **VAX**
3. *Memory* Size
  1. Moor's Law on Memory Growth, memory addressing
  2. Memory limitation of embedded system

*Semantic Gap*: Gap between High Level Language and  
Computer Architecture

# Re-look at Instruction set design-RISC

1980s : Ditzel & Patterson : *RISC (Reduced Instruction Set)*

Architecture

## *Research Outputs*

1. 1975 IBM 801 Project: Eckert Cocke .ECL based 24 bit registers not commercialized.
1. 1980 Berkeley : RISC-I and RISC –II: Patterson’s team, MOS based 32 bit registers. Targeted towards **Smalltalk and LISP.**
3. 1981 Stanford : MIPS Computer Hennessy published explanation of RISC advantages over VAX
4. 1986 : HP converted its Minicomputer to RISC (HP Precision Architecture)
5. 1987 : SUN SPRAC based on RISC-II
6. 1990 : IBM RISC RS 6000
7. 1998 : Alpha based MIPS 2000

# Evolution of Instruction Sets

- RISC ISA design philosophies:
  - Smaller is faster
    - small number of instructions
    - relatively small number of register types
  - Good design demands compromise
    - only a few instruction formats to handle special needs
  - Make the common case fast
    - most often executed instructions or heavily used features need to be optimized
  - Pipelining, single-cycle execution, compiler technology, etc.

# Instruction set Architecture

1. *Stack* All operations on a Stack. Operand Implicit
2. *Accumulator* One operand and Result on Accumulator
3. *Register-Memory* Operand on Register and on Memory
4. *Load-Store (RISC)* All Operand on explicit GPR

*We will use RISC MIPS Instructions in this class*

# Advantages & Disadvantages of different Instruction Architecture

Architecture	Advantages	Disadvantages
<i>Stack</i>	Simple encoding, Operand and result Location fixed	Operand must be correct order in the Stack. Operand need loading on stack
<i>Accumulator</i>	Simple instruction as only one Operand to be specified	Operation must be correct order to Have the correct operand on accumulator
<i>Register-Memory</i>	Least number of instruction	Complex instruction set. Decoding Is complex. Potential of variable Length instruction
<i>Load-Store</i>	Operand on Register, operand Can be used without additional Instruction.	Larger Instruction and encoding complexity

# Instruction set Architecture

<i>OPCODE</i>	<i>Operand-1</i>	<i>Operand-2</i>	<i>Operand-3</i>
---------------	------------------	------------------	------------------

**?** :      Where are these operands  
            What types of operands  
            How to get access to these operands

- 1. *Stack***                      All operations on a Stack. Operand Implicit
- 2. *Accumulator***              One operand and Result on Accumulator
- 3. *Register-Memory***      Operand on Register and on Memory
- 4. *Load-Store***                All Operand on explicit GPR

# Instruction Set Operations

- Typical machine instructions needed include:
  - **Data transfer** (reg-reg, reg-memory, memory-reg, ...)
  - **Arithmetic** (integer/floating point: add, subtract, multiply, ...)
  - **Logic and non-numeric** (Boolean, bit manipulation, string operations, etc.)
  - **Program control** (branches, jumps, PC manipulation, proc calls, ...)
  - **I/O operations**
  - **System operations** (OS calls, memory mngmnt, ...)

# Logical Operations

Logic : {True, False}  $\rightarrow$  {1,0}

**Bit String** can represent a logical statement

Operations of a Bit Strings

**Shift:** Move all the bits on left or right, filling the empty bit position with '0'.

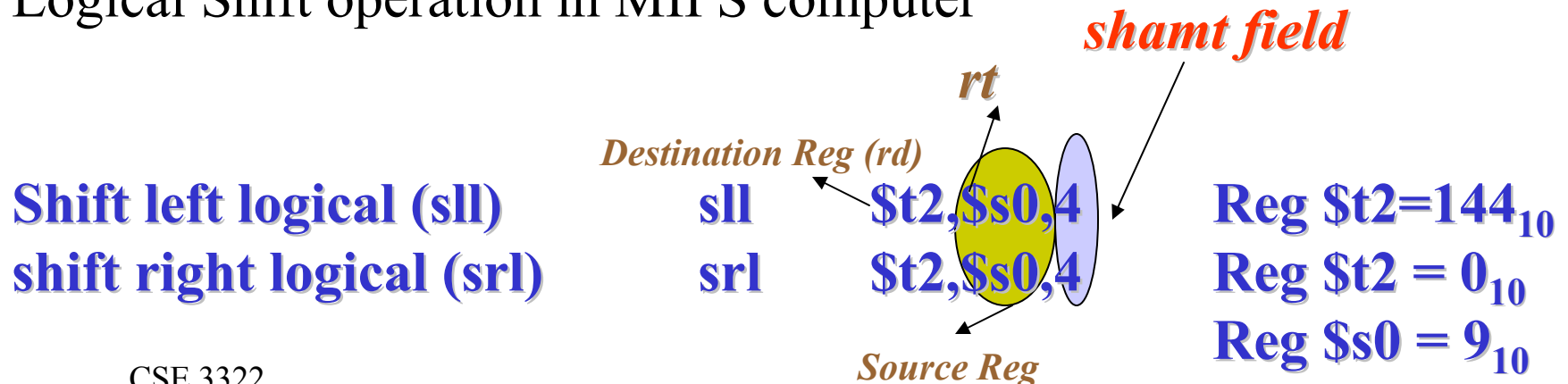
Consider a register  $\$s0$  of 8 bits contains decimal '9'

$$00001001_2 = 9_{10}$$

Shift left by 4 places *empty bit positions*

$$1001XXXX_2 = 10010000_2 = 144_{10}$$

Logical Shift operation in MIPS computer



# MIPS Instruction encoding

## Format type -R

<i>Op Code (6)</i>	<i>rs (5)</i>	<i>rt (5)</i>	<i>rd (5)</i>	<i>shamt(5)</i>	<i>function(6)</i>
--------------------	---------------	---------------	---------------	-----------------	--------------------

*add:*  $0_{10}$   $0_{10}$   $0_{10}$   $32_{10}$

*sub:*  $0_{10}$   $0_{10}$   $34_{10}$

## Format type -I

<i>Op Code (6)</i>	<i>rs (5)</i>	<i>rt (5)</i>	<i>address (16)</i>
--------------------	---------------	---------------	---------------------

*add(imm):*  $8_{10}$  *Constant*

*Load:*  $35_{10}$  *Constant*

*Store:*  $43_{10}$  *Constant*

# MIPS Registers

## 32 Registers:

Registers  $\$s0...\$s7$  maps to registers 16-23

Registers  $\$t0...\$t7$  maps to registers 8-15

Other registers are special registers like  $\$zero$ ,  $\$gp$ ,  $\$sp$ ,...

***add***  $\$s1, \$s2, \$s3$

$0_{10}$	$18_{10}$	$19_{10}$	$17_{10}$	$0_{10}$	$32_{10}$
----------	-----------	-----------	-----------	----------	-----------

***sub***  $\$s1, \$s2, \$s3$

$0_{10}$	$18_{10}$	$19_{10}$	$17_{10}$	$0_{10}$	$34_{10}$
----------	-----------	-----------	-----------	----------	-----------

***lw***  $\$s1, 100(\$s2)$

$35_{10}$	$18_{10}$	$17_{10}$	$100_{10}$
-----------	-----------	-----------	------------

$\$s1 = \text{Memory}[\$s2 + 100]$

# MIPS Memory

**Memory address Registers: 32 bits**

**Memory is arranged in Words of 4 bytes (32 bits)**

**Total memory words =  $2^{30}$**

**First word address: Memory[ $0_{10}$ ]**

**2<sup>nd</sup> word address: Memory[ $4_{10}$ ]**

.....

.....

.....

**Last word address: Memory[ $4294967291_{10}$ ]**

# Logical Operations

Instruction	C Program	JAVA	MIPS	
shift left	<<	<<	sll	
shift right	>>	>>>	srl	
Bit by Bit AND	&	&	and, andl	
Bit by Bit OR	/	/	or, orl	
Bit by bit NOT	~	~	nor	

## Logical Operation

and \$t0, \$t1, \$t2

Reg \$t0 = Reg \$t1 & Reg \$t2

andi \$s1, \$s2, 100<sub>10</sub>

Reg \$s1 = Reg \$s2 & 100<sub>10</sub> in bits

or \$t0, \$t1, \$t2

Reg \$t0 = Reg \$t1 / Reg \$t2

nor \$t0, \$t1, \$t3

Reg \$t0 = ~(Reg \$t1 / Reg \$t3)

## NOR = NOT OR

contains all '0'



$$A \text{ NOR } 0 = \text{NOT} (A \text{ OR } 0) = \text{NOT}(A)$$

$$A=0 \quad \quad \quad = \text{NOT} (0 \text{ OR } 0) = \text{NOT}(0) = 1$$

$$A=1 \quad \quad \quad = \text{NOT} (1 \text{ OR } 0) = \text{NOT}(1) = 0$$

# Logical Operations in MIPS

***and*** \$s1, \$s2, \$s3

$0_{10}$	$18_{10}$	$19_{10}$	$17_{10}$	$0_{10}$	$36_{10}$
----------	-----------	-----------	-----------	----------	-----------

***andi*** \$s1, \$s2, 100

$12_{10}$	$18_{10}$	$17_{10}$	$100_{10}$
-----------	-----------	-----------	------------

***sll*** \$s1, \$s2, 10

$0_{10}$	$0_{10}$	$18_{10}$	$17_{10}$	$10_{10}$	$0_{10}$
----------	----------	-----------	-----------	-----------	----------

***srl*** \$s1, \$s2, 10

$0_{10}$	$0_{10}$	$18_{10}$	$17_{10}$	$10_{10}$	$2_{10}$
----------	----------	-----------	-----------	-----------	----------

# Control Flow Instructions

## *Standard Techniques*

1. Jump
2. Branch
3. Procedure Call
4. Procedure returns

## *PC-relative*

Simple, only displacement bits are needed

Position Independence

Fewer bits are required, as jumps normally close to PC

## *Register indirect jumps*

When target is not known at compile time

Case or Switch statements

Virtual functions or Methods

Function Pointers

Dynamically shared Library

# Branch Condition Evaluation

*Condition Code (CC)*

*Condition Register (CR)*

*Compare and Branch (CB)*

Name	Example	Test	Advantage	Disadvantage
<i>CC</i>	80X86,SPARC PowerPC	Test special bits set by ALU, possibly by program	Some time condition is set free	CC is extra state Constrain the inst ordering
<i>CR</i>	Alpha,MIPS	Test arbitrary registers with result of a comparison	Simple	Uses up registers
<i>CB</i>	VAX,PA-RISC	Compare is part of branch. Often compare is limited subset.	One instruction rather than two For a branch	Too much work for pipeline architecture

# Decision Instruction

1. *go to* statement: unconditional Jump

MIPS used for unconditional *Jump* the Jump instruction.

*j*      *Exit (address)*



2. *if* Statement: Conditional Jump

MIPS used two assembly language instruction like

*if < Condition of Jump=True > then go to address”*



*beq*      *Register1, Register2, L1*      *Jump to L1 when Reg1 = Reg2*

*bne*      *Register1, Register2, L1*      *Jump to L1 when Reg1 != Reg2*

# Logical Operations in MIPS

**beq** \$s1, \$s2, 100

$4_{10}$	$17_{10}$	$18_{10}$	$25_{10}$
----------	-----------	-----------	-----------

**bne** \$s1, \$s2, 100

$5_{10}$	$17_{10}$	$18_{10}$	$25_{10}$
----------	-----------	-----------	-----------

**j** 1000

$2_{10}$	$250_{10}$
----------	------------

**slt** \$s1, \$s2, \$s3    if (\$s2 < \$s3) then \$s1 = 1

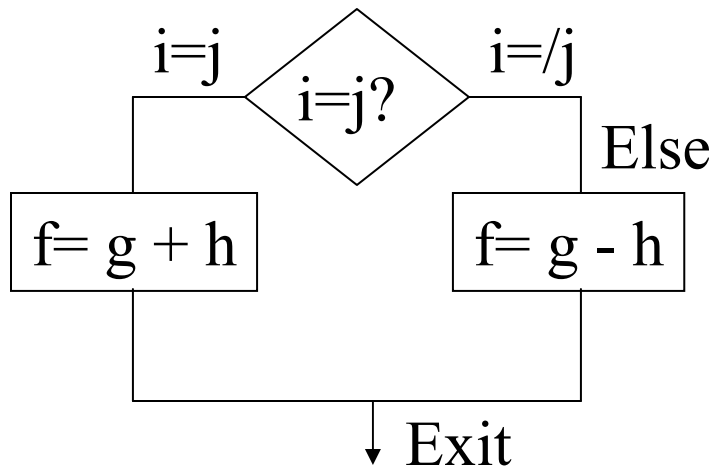
$0_{10}$	$18_{10}$	$19_{10}$	$17_{10}$	$0_{10}$	$42_{10}$
----------	-----------	-----------	-----------	----------	-----------

# if-then-else instruction

C *'if-then-else'* statement

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

f = Reg \$s0  
g = Reg \$s1  
h = Reg \$s2  
i = Reg \$s3  
j = Reg \$s4



## MIPS instruction

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

# Loops

## C Loop Statement

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

i = Reg \$s3

k = Reg \$s5

**Base Register** of Array Save [ ] = Reg \$s6

\$t1 & \$t0 : Temp reg

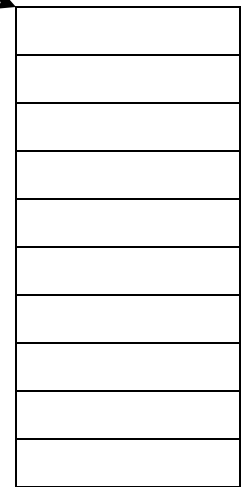
Reg \$s6

### MIPS Code

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

Exit:

Byte addressing  
4 bytes per word



Array save[ ]

**Basic Block** : sequences of instructions without branches

# Loops

Test of variable for less than another variable

MIPS compare two registers and set another register to '1' if the first is less than the second register, other wise the third register is '0'.

*set on less than (slt)*

## MIPS Code

*slt \$t0, \$s3, \$s4*

*Reg \$t0 = 1 if \$s3 < \$s4*

*Reg \$t0 = 0 if \$s3 >= \$s4*

*slti \$t0, \$s2, 10*

*immediate*

*Reg \$t0 = 1 if \$s2 < 10*

*Reg \$t0 = 0 if \$s2 >= 10*

*Reg \$zero = Contains '0'*

# Case

**Case/Switch Instruction** : Multiple Jump destinations based on value of a single variable.

Implementation.

Chain of if-then-else clauses

Jump address Table

MIPS Support

**Jump Register instruction**: Jump to the instruction, the address of which is at a register.

# Procedure Call

1. *Caller Saving* : Calling procedure must save registers that it must have after completion of the procedure call.
2. *Callee Saving*: The called procedure will save the registers and restore them back when the control is returned.

# Procedure Call

*Six steps* of Procedure/Function execution

1. *Parameter passing* to procedure
2. Program *control transfer* to procedure
3. Acquire the storage resources needed for procedure
4. Perform task
5. *Results transfer* to main program
6. *Return address control*

MIPS Registers for Procedure Call

*\$a0-\$a3*: 4 registers for parameter passing

*\$v0-\$v1*: 2 Registers for result transfer

*\$ra*: Return address register

MIPS special Instruction for Procedure Call

*jal* *jump and link*: Jump to the address in the instruction and save the *address of the following instruction* in *\$ra*.

*jal* *Procedure Address*

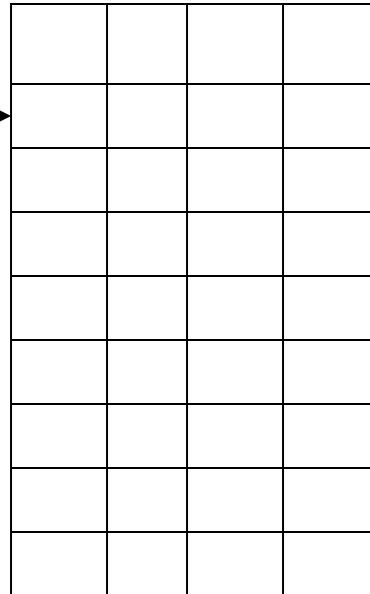
# Procedures Call

*jal Procedure Address*

*PC : Program Counter*

To keep track of the address of the instruction, a counter is maintained. This counter is incremented by the word size of the instruction to get the address of the next instruction.

PC



*Next Inst. Address*

*PC=PC+4*

*Jump Register instruction*

*jr \$ra*

*PC=\$ra*

Program store

# Procedure Call

## MIPS Procedure Call

1. *\$a0 - \$a3*: loaded with procedure parameters
2. *jal X*  
\$ra = PC + 4  
PC = Procedure address
3. Procedure frees up the registers that is used for procedure execution
4. Procedure execution.
5. *\$v0-\$v1*: Procedure results transferred
6. *jr \$ra*  
PC = \$ra

# Stack and Procedure Call

Data structure *Stack* is used when we need more registers than what is available in MIPS for procedure execution (*point-3*).

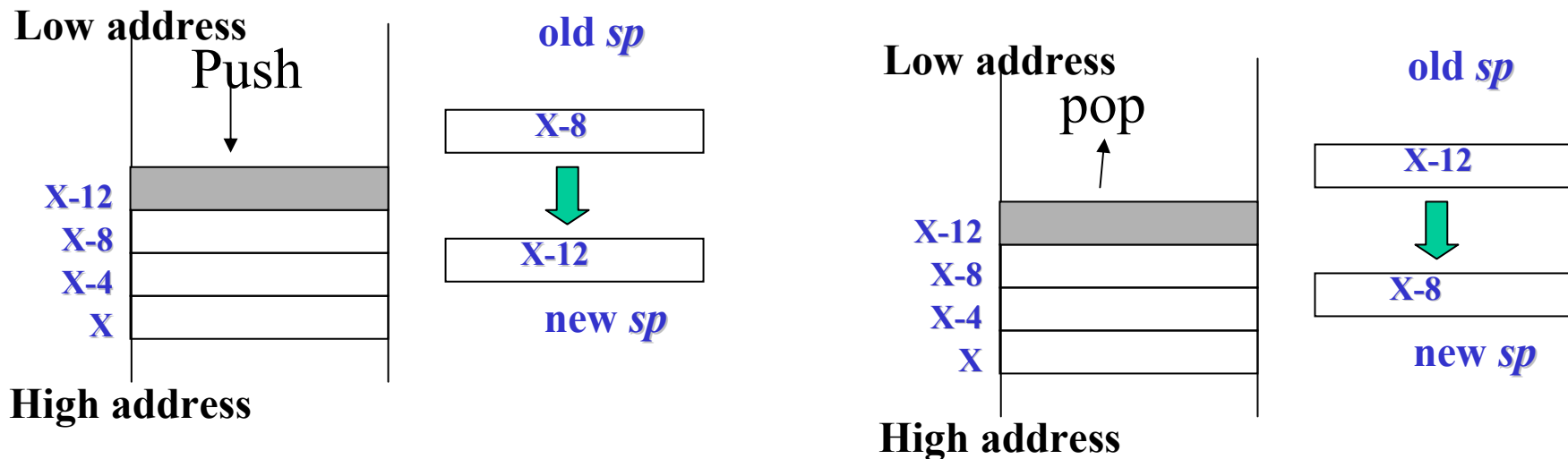
**Stack** : a last in first out (LIFO) queue

**Stack Pointer**: most recently allocated address in the stack. MIPS stack pointer is  $\$sp$ .

**push**: placing data into stack.

**pop**: removing data from the stack

*Stack address grows from higher address to lower address*



# Register Saving during Procedure Call

## C Procedure call

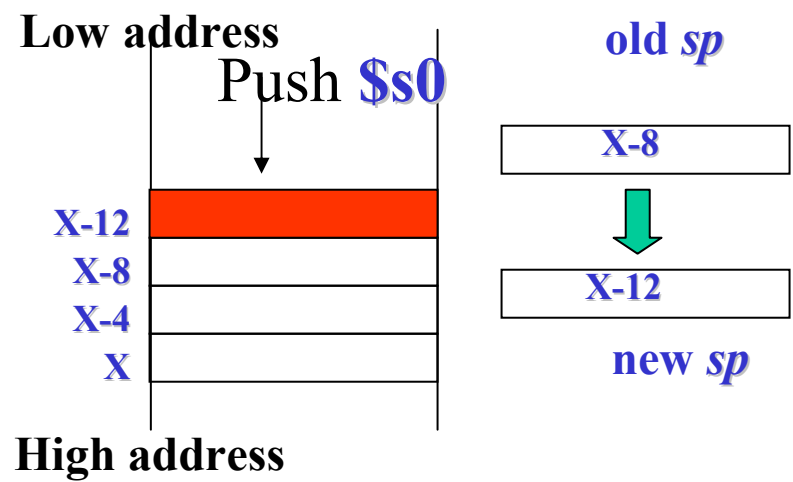
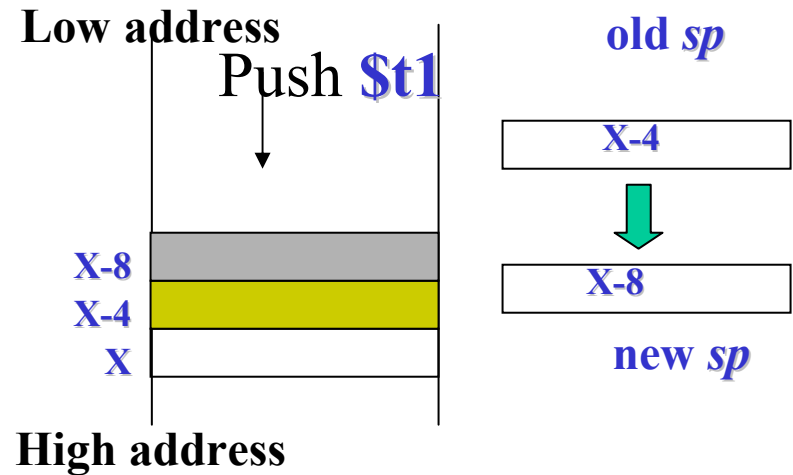
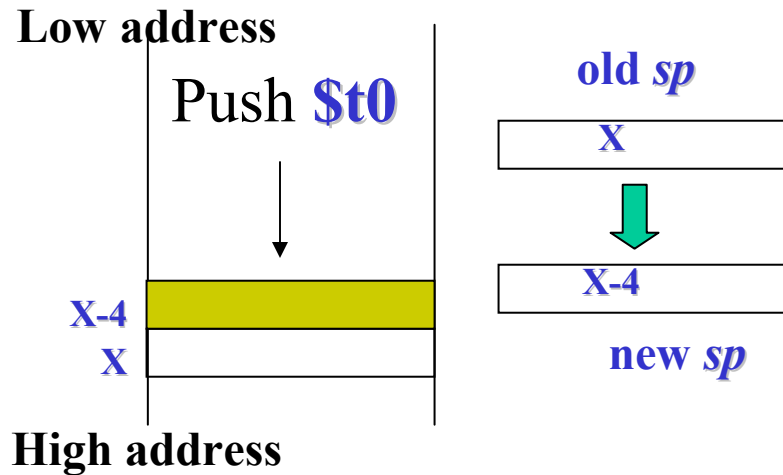
```
int    leaf_example (int g, int h, int i, int j)
{
    int f;

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

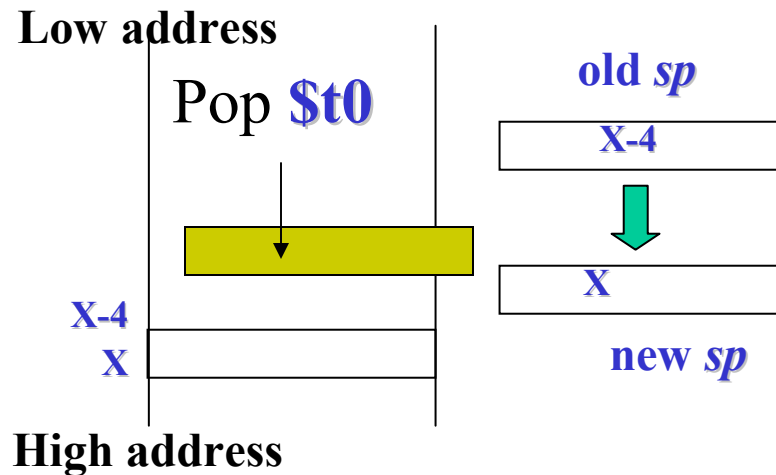
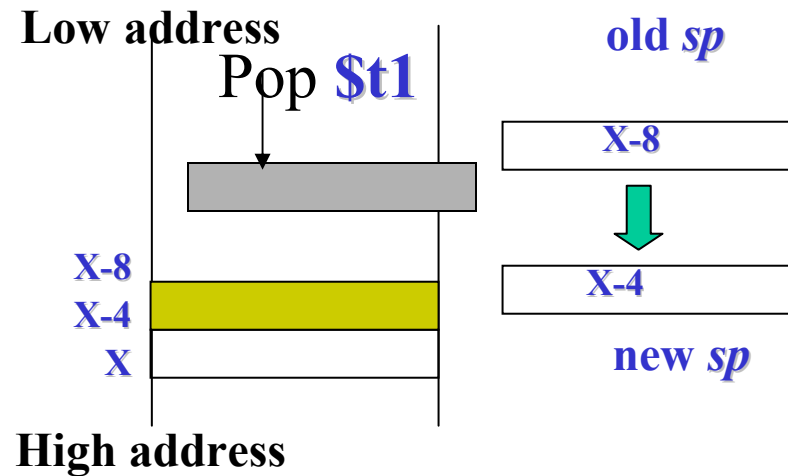
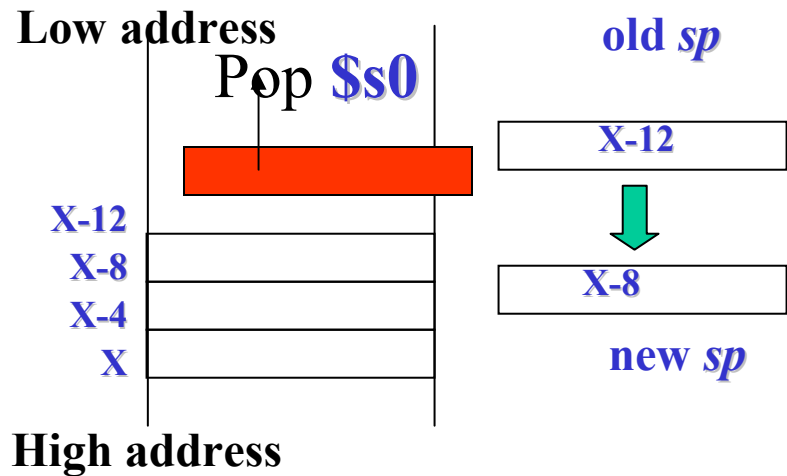
The values of *g*, *h*, *i*, *j* are stored in argument registers **\$a0 ..\$a3**.  
The result *f* is on register **\$v0**.

To execute the procedure we need save registers **\$to, \$t1 and \$s0**.  
So before we start executing the procedure we have to free up the registers \$t0, \$t1 and \$s0. We use **stack** to free up the registers.

# Register Saving (PUSH) during Procedure Call



# Register pop during Return



# Nested Procedure Call

Example page 83

# Allocating Space in Stack

The following items are stored in the stack for a procedure. The whole area is called **procedure frame** or **activation record**.

- argument registers (if any)

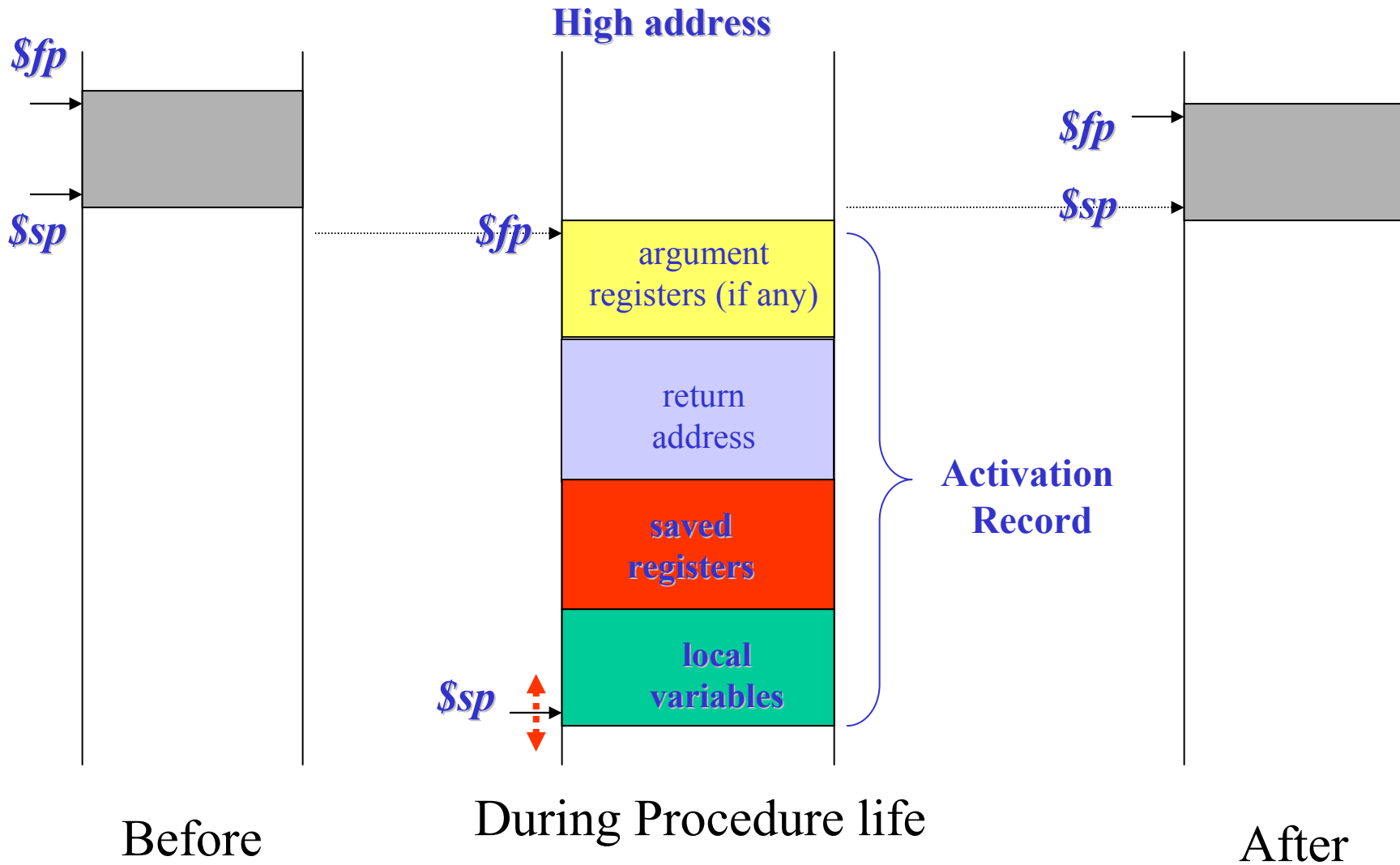
- return address

- save saved registers

- local variables of the procedure like array, data structure.

If stack pointer is used to refer the local variables, there is continuous need to make the address offset computation. Stack pointer changes during the procedure life. To avoid that a **fixed base register** is used for local variable addressing, called **Frame Pointer (\$fp)**

# Add Activation Record



# Memory Allocation

**Stack:** local data

different register savings for procedure calls.

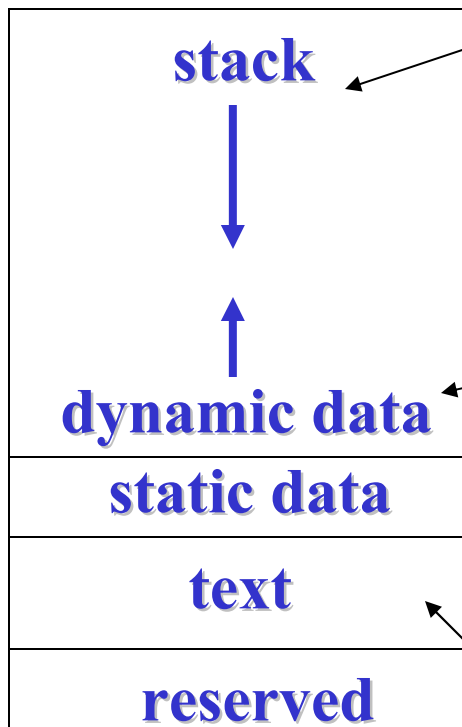
**Heap:** Static variables : constants

Dynamic data structures.

Highest address:  $7fff\ fffC_{hex}$

**free() ; releases space  
from stack**

Memory Management  
Memory Leak  
Dangling Pointers



**Heap: Link list  
malloc() : allocates space  
in heap**

$\$gp\ 1000\ 8000_{hex}$

$1000\ 0000_{hex}$

$pc\ 0040\ 0000_{hex}$

**Machine Code**

Lowest address:  $0000\ 0000_{hex}$

# strings copy in C

Example Page 92

# JAVA Character and String

**Unicode** : Universal encoding of Human Languages, Unicode has as many alphabets as there are symbols in ASCII.

*JAVA uses 16 bits to represent a character.*

16 bits = MIPS half words

MIPS instructions of half-words

*lh : Load half word*: Take half word and place it on right 16 bit position of the register.

*sh: Store half word*: Store the right most 16 bits of register in the memory.

A character variable allocated to stack occupies 4 bytes position to *align* the lh and sh instructions.

Java string variable stores 2 half words per word,

C language string variable stores 4 bytes per word.

# Interpreting Address

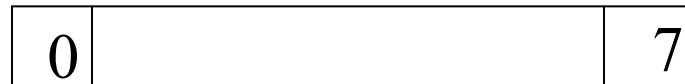
## 1. *Little Endian*

(PDP-11, Intel 80x86)



## 2. *Big Endian*

(IBM360/370, Motorola)



## *Alignment*

Byte Address :  $A$

Object size :  $s$

Byte address oriented memory will align if  $Mod[A, s] = 0$

## *Addressing Mode*

Effective Address

PC – Relative addressing: Mainly used for control transfer



# Addressing in Branches and Jumps

## JUMP Instruction

*j*      *address (26 bits)*

Word addressing not Byte. This makes the address range to  $2^{28}$ . But the complete range of PC is  $2^{32}$ . The 4 upper bits come from the PC, so Jump instruction replaces only lower 28 bits of PC.

## Branch Instruction

*bne*    *\$s0, \$s1, Address (16 bits)*

Maximum Memory address covered  $2^{16}$ . This is very low value.

How we can extend it to  $2^{32}$ .?

## PC Relative addressing.

In this case, the address immediate bits are added to PC to find the instruction location. As MIPS use Byte addressing, the scope of the branch is increased by considering the *immediate field value is in words*. As PC is already changed before the completion of branch instruction to  $PC = PC + 4$ , the relative address is with respect to new PC value.

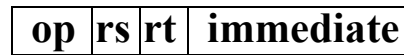
# Branch and Jumps

Example Page 98 and 99

# Memory Addressing

## *Addressing Modes*

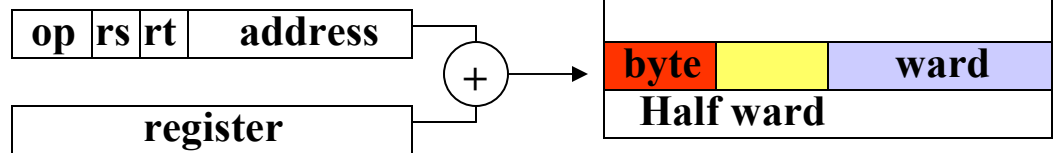
Immediate



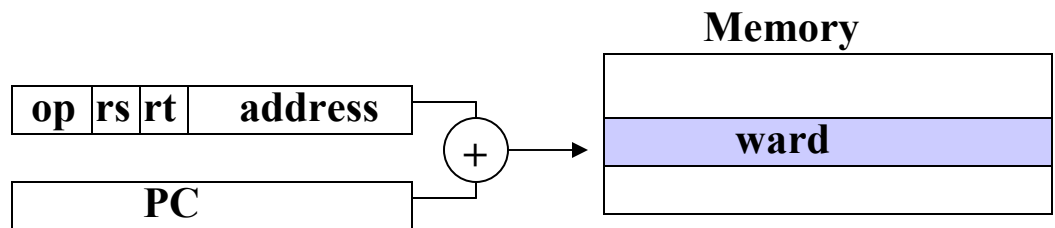
Register



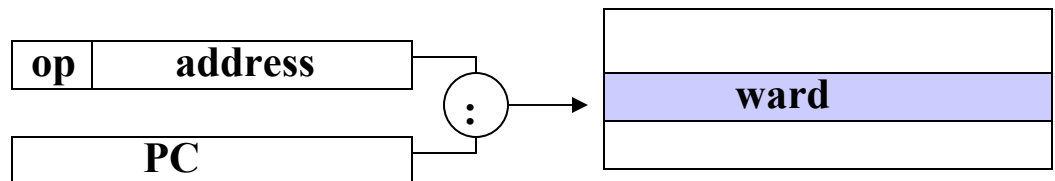
Displacement



PC Relative



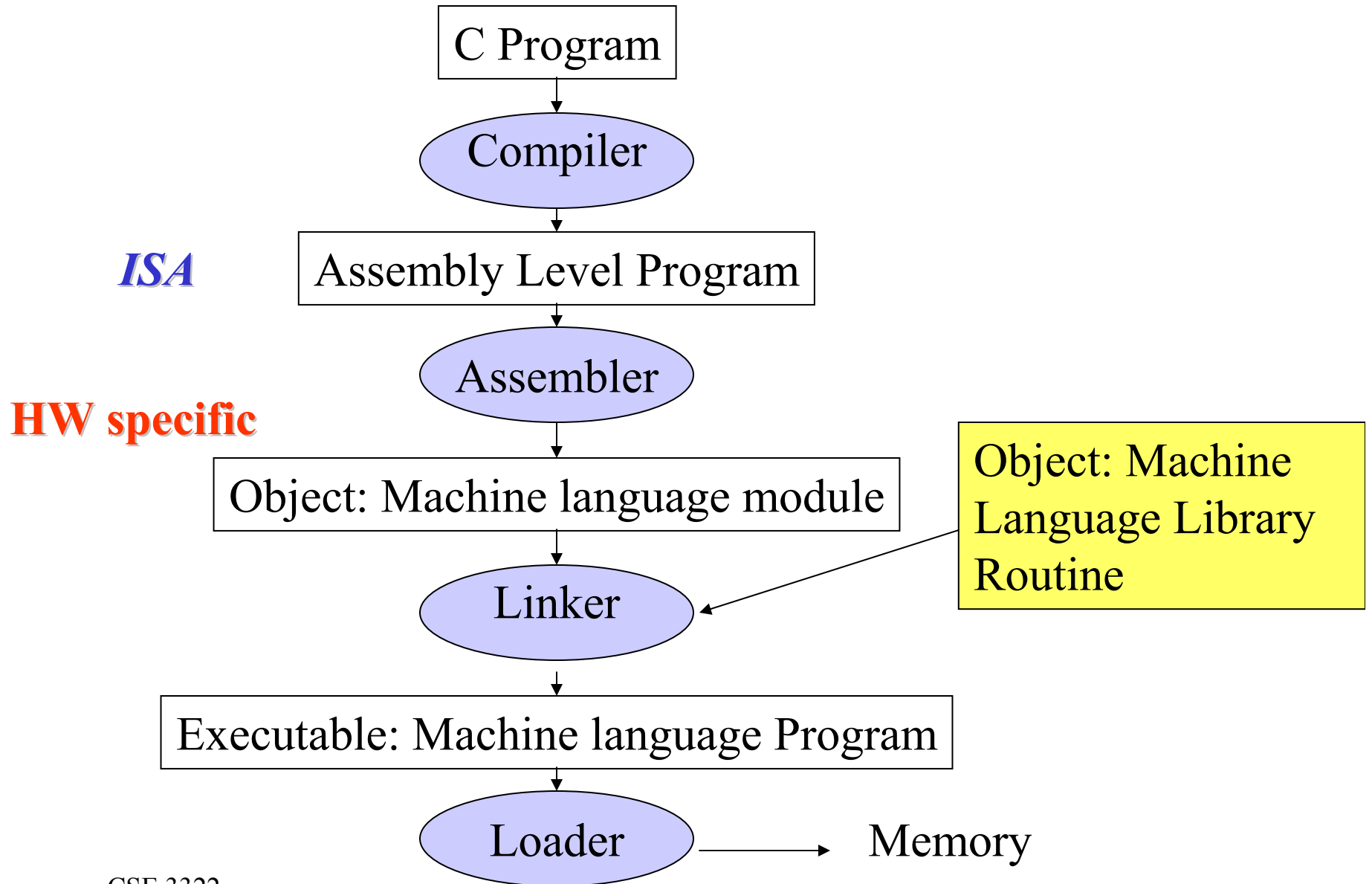
Pseudo direct



# Decoding Machine Code

Figure 2.25 and 2.26

# C Program Translation



# Compiler & Assembler

**Compiler:** Converts C program to Assembly language program

**Assembler:** Compiler creates Assembly level **Pseudo instructions** those are very close to HW instruction. Assembler converts the Pseudo instruction to **machine language equivalent**. Normally Pseudo instruction is richer than HW machine instruction, so Assembler has to convert those to multiple machine language instruction. It converts the different numbering system to one base like **Hexadecimal in MIPS**.

Assembler creates the **object file** that includes

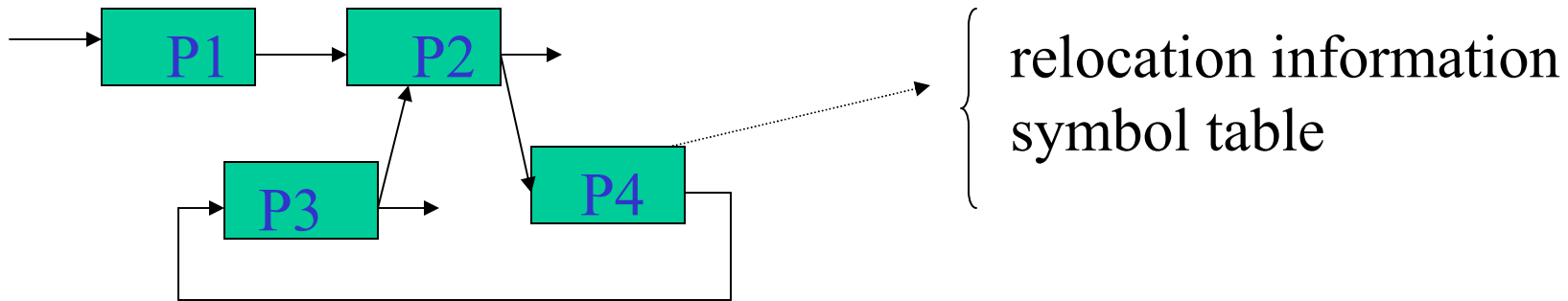
1. Machine language program
2. Data
3. information required to put it on memory like symbol table.

**Symbol table** contains symbols and addresses.

# UNIX Object File

1. ***Object file Header***: size and position of other pieces of object file
2. ***text segment***: machine language code
3. ***static data***: static data for life of the program and dynamic data that shrinks and grows as needed.
4. ***relocation information***: instruction and data that depends on absolute memory address.
5. ***symbol table***: labels that are not defined. external labels.
6. ***debugging information***: module compilation

# Linker or Link editor



All external references to be resolved.

- Place Code and Data modules symbolically in memory.
- Determine address of data and instructions
- Patch internal and external references
- Produces the **executable code**

**It is faster to patch code than recompile and reassemble.**

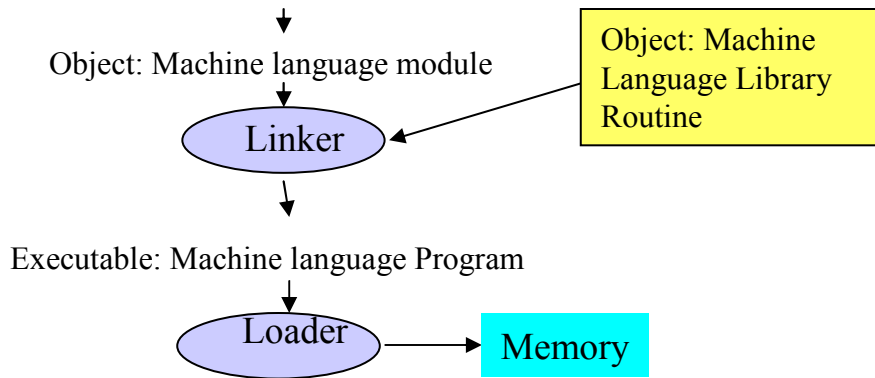
# Linker

Example Page 109: Linking object Files

# Loader

1. Executable file is in a disk like CD.
2. Operating system uses the Loader program to read this executable file and stores in the memory and then start the execution. The steps in UNIX operating system
  - a. Read the Header section and determine the text and data segments.
  - b. Create an address space sufficient for the text and data.
  - c. copies the instructions and data from executable file into memory.
  - d. Copies the parameters, if any to the stack.
  - e. initialize the machine registers and stack pointers.
  - f. Jumps to a start up routine that copies the parameters to the argument registers and calls the main routine.

# Dynamically Linked Libraries



Library Routine is **statically linked**

- *Part of executable code*
- *Version management* difficulties
- *Large size* of the library. 2.5 MB for UNIX Library

**DLL** : Library Routines are **linked at the RUN time**.

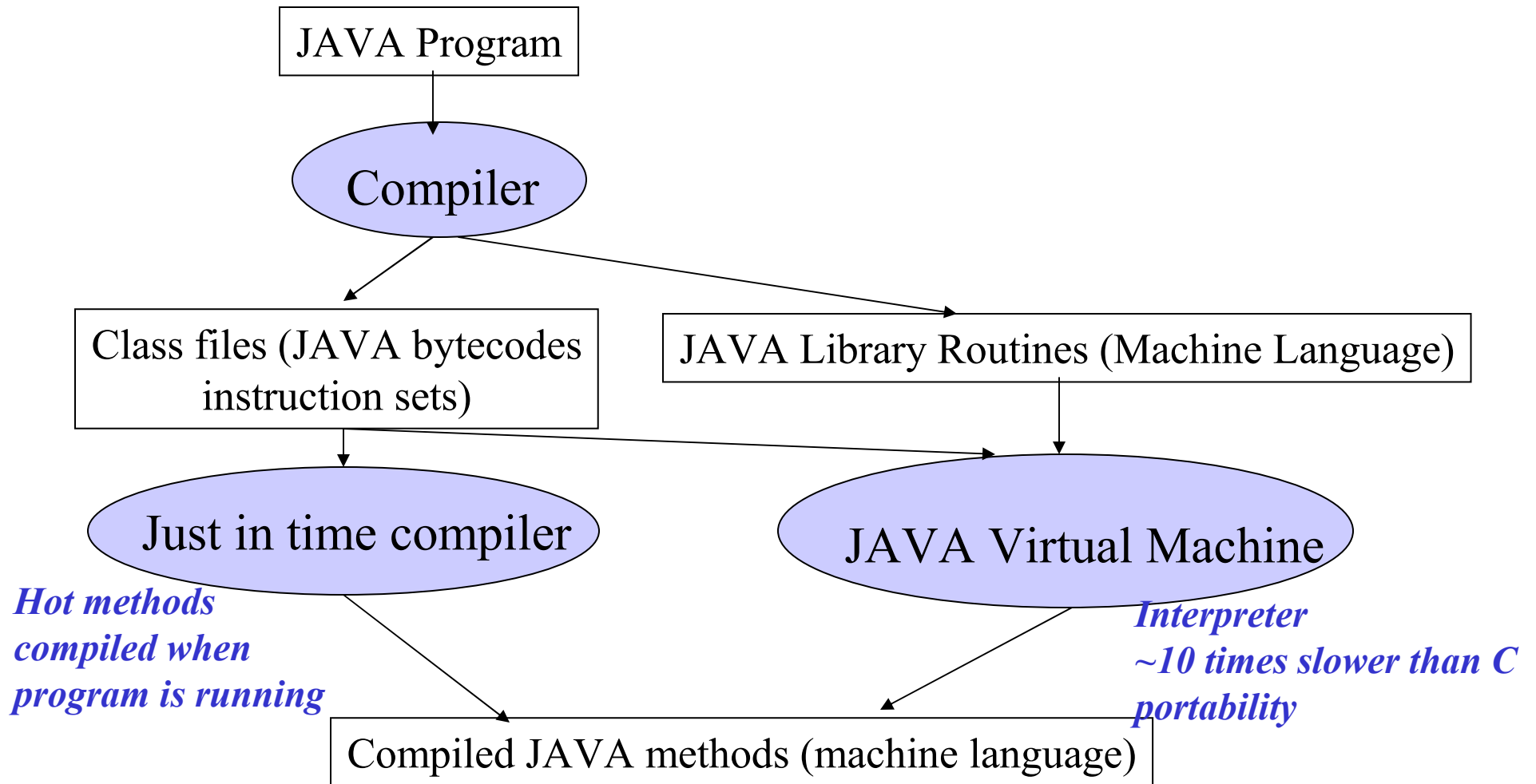
Both program and DLL keeps *extra information on the location of normal procedures and their names*.

1. **Early version**: During initial run of DLL, the loader run a dynamic Linker. This uses the extra information and find the appropriate libraries and update external references.
2. **Lazy** DLL: Each procedure is linked only after it is called.

# Lazy DLL

Figure-2.29

# Starting JAVA Program



# Role of Compiler

1. Compiler functions
2. Register allocation
3. Processor dependent optimization
4. Stack for local variable
5. Heap for dynamic objects
6. Assembly vs. compiled language

# Compiler Functions

## ***Objective of Compiler:***

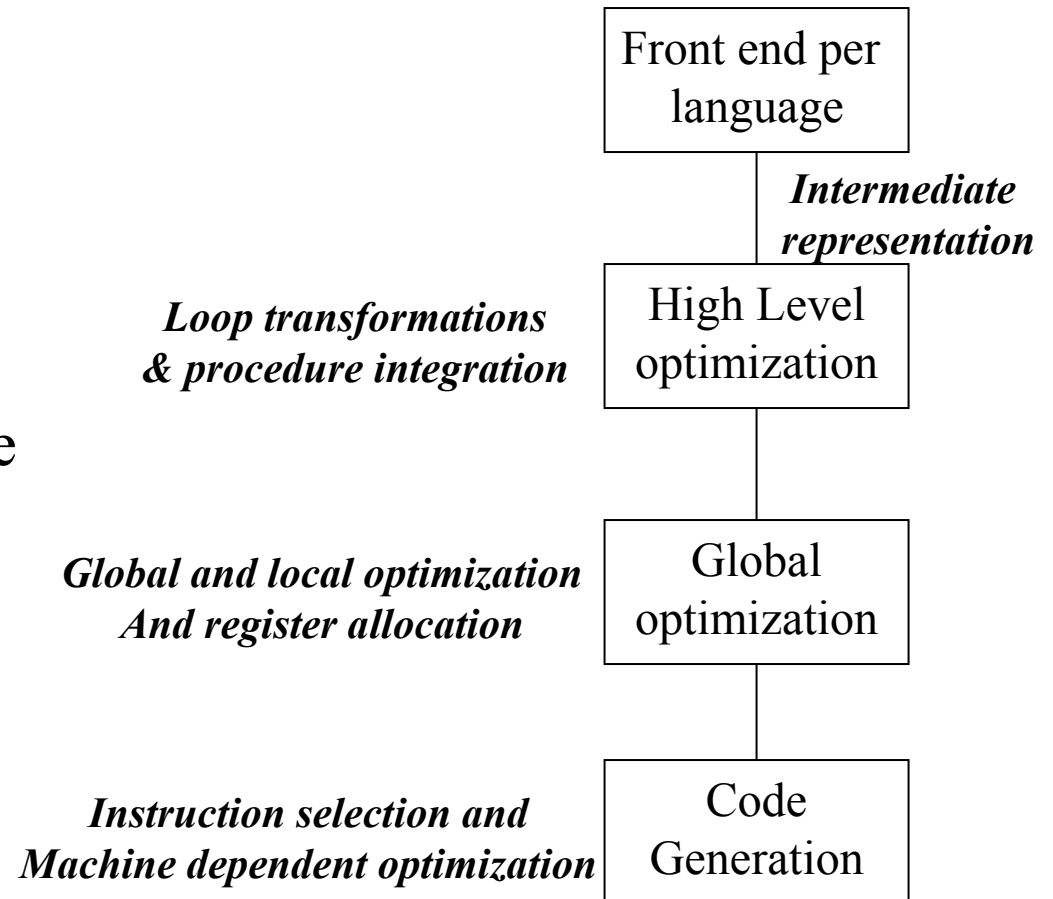
*Correctness*  
*speed of execution*  
*fast compilation*  
*debugging support*  
*interoperability*

Area of Interest for Architecture

***Optimization***

***Code Generation***

***Compiler Pass:*** *One phase in which  
The compiler reads and transforms  
the entire program.*



# Register Allocation

Problem of allocating Registers to instructions.

Instructions require Registers.

Graph Coloring Problem

1. Instructions that can not share the registers are viewed as adjacent nodes.
2. Create a inference graph of the instructions
3. Color the Graph with minimum number of colors.

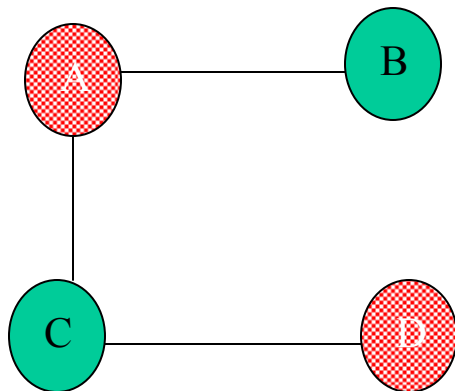
Graph coloring is NP –complete problem

There are heuristic algorithms that can work well in practice.

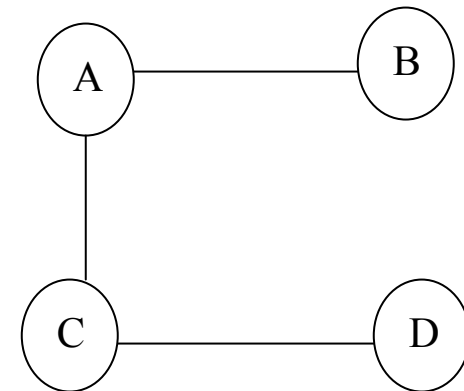
# Graph Coloring

## Program Segment

A=  
B=  
....  
....  
..B...  
C=  
...A..  
D=  
...D..  
...C..



**Colored Graph**



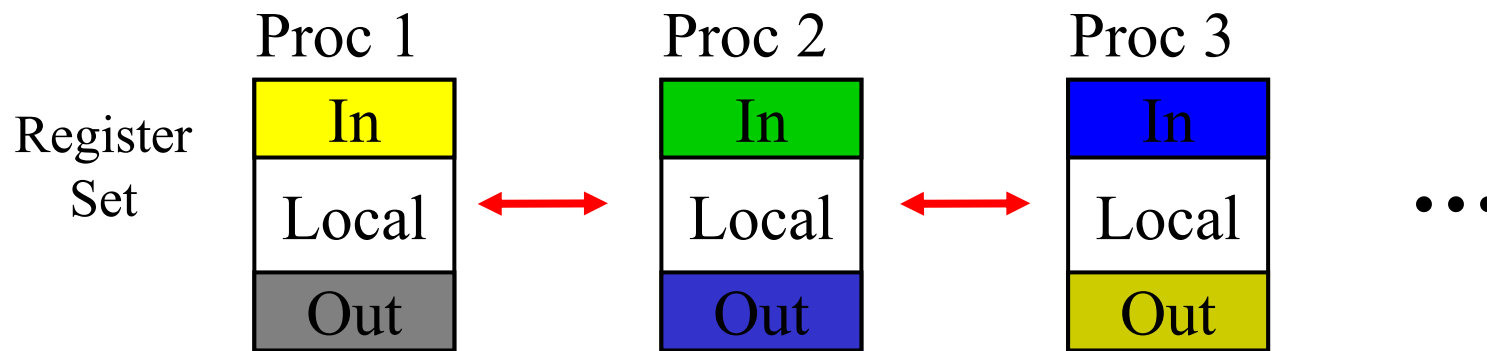
**Inference Graph**

## Register Allocation Code

R1=  
R2=  
....  
....  
..R2...  
R2=  
...R1..  
R1=  
...R1..  
...R2..

# Procedure/Function Calls and Register Windows

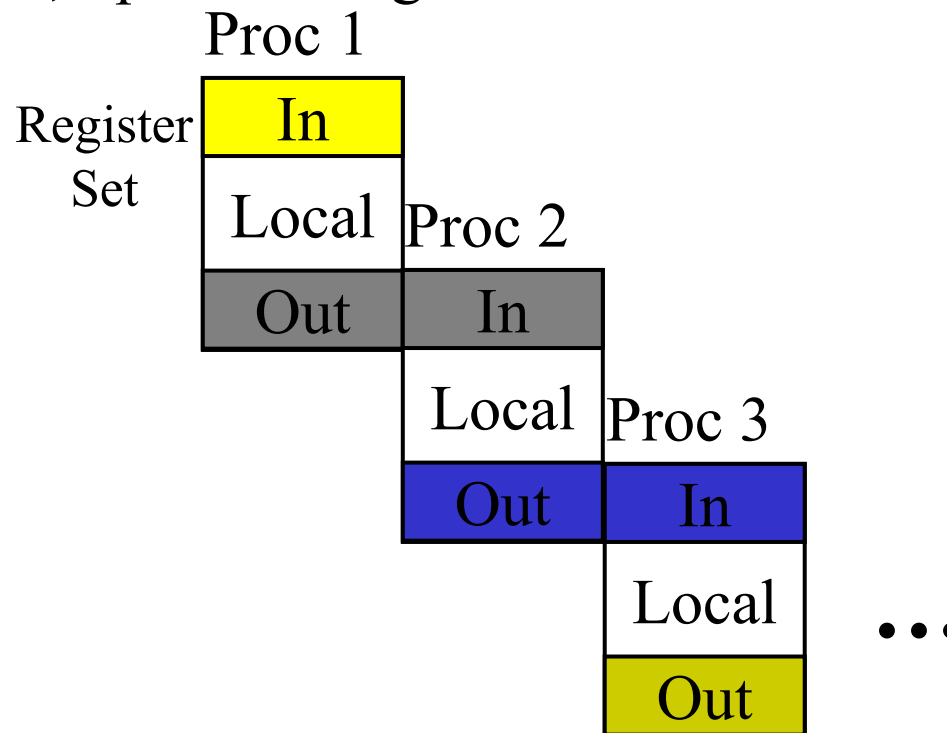
- Procedure/function calls are time-consuming due to:
  - Saving/restoring registers on each call/return
  - Passing parameters and results to/from the procedures



- Typically, procedures are called by other procedures.
- Rather than save **all** the state from one call to the next the registers can be re-used between calls.
  - Reduce the size of state that must be saved.
  - More efficiently utilize registers.

# Procedure/Function Calls and Register Windows

- Hardware support for **overlapping register windows**.
  - Example, Sparc-V8 register windows.



# Optimization Types

Optimization Name	Explanation	% of Total Number of optimizing transforms
<b>High Level</b>	At or near the source level: Processor independent	NM
Procedure Integration	Replace procedure call by procedure body	
<b>Local</b>	Within straight line code	
Common expression elimination	Replace two instances of same computation by single copy	18%
Constant Propagation	Replace all instances of variable that is assigned a constant with a constant	22%
Stack height reduction	Rearrange expression tree to minimize resources needed for expression evaluation	NM
<b>Global</b>	Across a branch	
Eliminate common sub expression	Same as local, but this version crosses branches	13%
Copy Propagation	Replace all instances of variable A that has been assigned X with X.	11%
Code Motion	Remove code from a loop that computes same value on each iteration of the loop.	16%
Induction variable elimination	Eliminate array addressing calculations within loop	2%
<b>Processor dependent</b>	Depends on processor architecture knowledge	
Strength reduction	Ex: replace multiply by a constant with add and shift	NM
Pipeline scheduling	Reorder instruction	NM
Branch offset optimization	Select shortest displacement	NM

# Impact of optimization on Instruction Count

Optimizations	% Faster
Procedure Integration	10
Local optimization only	5
Local optimization+Register allocation	26
Global + Local optimization	14
Global+Local Optimization and Register Allocation	63
Local + Global optimization + Register allocation and Procedure Integration.	81

*Measurement on 12 small FORTRAN and Pascal programs (Chow,1983)*

# Impact of Compiler Technology on Architecture

**Question:** How are variables allocated and addressed? How many Registers are required to allocate variables appropriately?

**Stack:** Used to allocate local variables, The stack is grown and shrunk on procedure call or return. Object on the stack are addressed relative to the stack pointers. Object are scalars and not vectors. Stacks are kept for activation records and not for computing expression.

**Global data area:** Used to allocate statically declared objects, such as global variables or constants. Most of these objects are array or data structures.

**Heap:** Used to allocate dynamic data objects that do not adhere to a stack discipline. Objects in the heap are accessed with pointers.

*Register allocation is much more effective with stack allocated objects.*

# How Compiler Works: C Example

Page 122-128

# Array Versus Pointers

# Intel IA-32 Instructions

**1978:** *Intel 8086* 16 bit architecture, extension of 8 bit 8080 processor  
Registers are dedicated (not GR).

**1980:** *Intel 8087* add floating point coprocessor with 60 floating point instructions. It uses stack processing.

**1982:** *Intel 80286* extend address space to 24 bits and elaborate memory protection scheme and memory mapping.

**1985:** *Intel 80386* extend 80286 to 32 bits registers and address space. The registers are GP and supports paging and segmentation.

**1989-95:** *Intel 80486* ('89), *Pentium* ('92) and *Pentium-Pro* ('95). Added 4 instructions to support multi processing.

**1997:** Intel Pentium and Pentium Pro added with *MMX*. 57 new instructions with floating point stack with SIMD.

**1999:** *Intel Pentium III* with 70 instructions as *SSE* (Streaming SIMD Extension). Add 8 separate registers with 128 bits width, single precision floating point. 4 32 bit floating point operations in parallel. Cache pre-fetch and streaming storage instruction.

# Intel IA-32 Instructions

**2001:** *Intel Pentium 4 SSE2* with 144 new instruction and floating point with double precision using 64 bit registers. 2 double precision floating point instructions in parallel. 144 instructions are existing MMX and SSE instructions in 64 bits.

**2003:** AMD enhance the IA-32 architecture to 64 bits address space. *AMD64* widens all registers to 64 bits. Increase registers to 16 and 128 SSE bits registers to 16. New instruction mode called long mode with 64 bit instructions. Add new prefix to instructions.

**2004:** Intel adopts AMD64 architecture labeled it as *EM64T* (Extended Memory 64 Technology). Intel added 128 bit atomic compare and swap mode. Included 13 additional instruction for SSE3 upgrade. This provides complex arithmetic and graphic operations on arrays of structures, video encoding, floating point conversion, and thread synchronization.