

# Instructions: Language of the Machine

# Instructions

- Language of the machine
- Its vocabulary is called instruction set.
- More primitive than higher level languages
  - e.g., no sophisticated control flow
- Less powerful operations
- We'll be working with the “MIPS” ISA
  - similar to other architectures developed since the 1980's
  - used by Cisco, NEC, Nintendo, SG, Sony, TI, Toshiba, ATI Technologies
  - 100 million of MIPS manufactured in 2002

# Instruction Categories

1. Information flow (data transfer)
  - Load from memory
  - Store in memory
2. Logic operations
  - AND, OR, EXOR
  - Shift, Negation
3. Arithmetic operations
  - Addition/subtraction
  - Multiplication
4. **Branch** operations (control flow)

# Instruction Types

- Instruction with different number of operands

## 1. Instruction with one operand

- `jump address`
- `jump $Register No`

## 2. Instruction with two operands

- `multiply $R2, $R3` ( $R2 = R2 * R3$ )

## 3. Instruction with three operands

- `add a, b, c` ( $a = b + c$ )
- `sub a, b, c` ( $a = b - c$ )
- `add a, a, b` ( $a = a + b$ )

# MIPS Arithmetic

- All arithmetic/logic instructions have 3 operands
- Operand order is fixed (destination first)
- Example:

C/C++ code:         $A = B + C$

MIPS code: `add $s0, $s1, $s2`

Note: `$sx` registers are associated with local variables by compiler

# First Design Principle

- Design Principle: *simplicity favors regularity*.
- Simplicity may complicate things as well.

C code:        `A = B + C + D + E;`

- Suppose registers `$s0`, `$s1`, `$s2`, `$s3`, `$s4` contain the values of `A`, `B`, `C`, `D`, `E`, respectively.
- MIPS code:  
          `add $t0, $s1, $s2`  
          `add $t1, $t0, $s3`  
          `add $s0, $t1, $s4`

# Second Design Principle

- In MIPS, 32 32-bit registers provided
  - Operands to all arithmetic/logic operations come from registers,
- Design Principle: *smaller is faster*.
- A very large number of registers would increase the clock cycle time
- Optimum number
  - the programmer's need for more registers
  - the designer's desire to keep the clock cycle short.

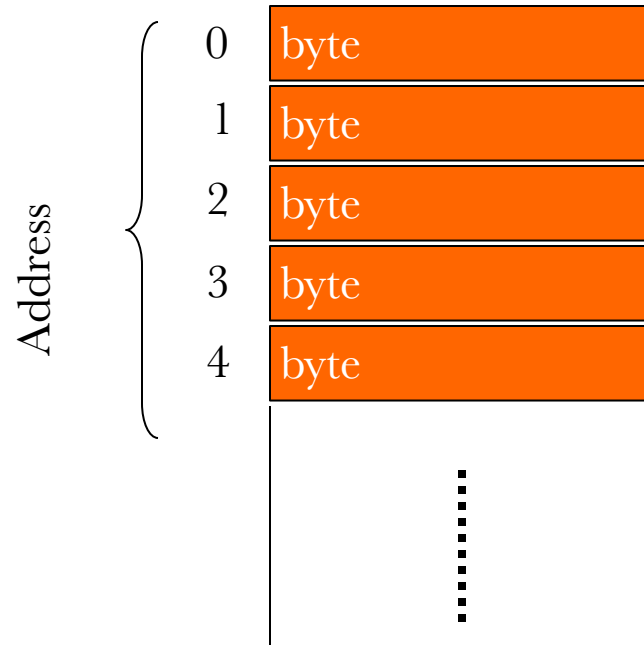
# Registers vs. Memory

- Operands of arithmetic/logic instructions must come from registers,
  - Computers have limited number of registers
- Compilers associate variables with registers
- Typical programs use more variables than the number of registers
  - For example, long arrays are kept in memory
- Thus, instructions for transferring data between memory and registers are needed.
- Data transfer instructions.



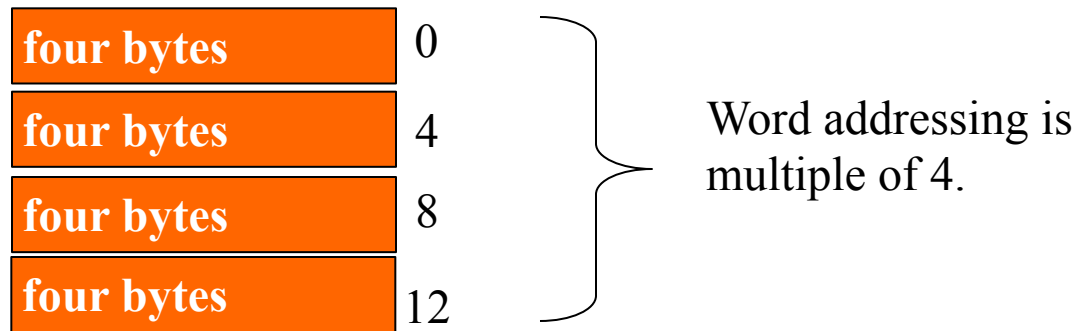
# Memory Organization

- Viewed as a large, single-dimensional array, with an address.
- A memory address is an index into this array
- To access to data in memory, instructions must supply the address
- "Byte addressing" means that the index points to a byte of memory.



# Memory Organization

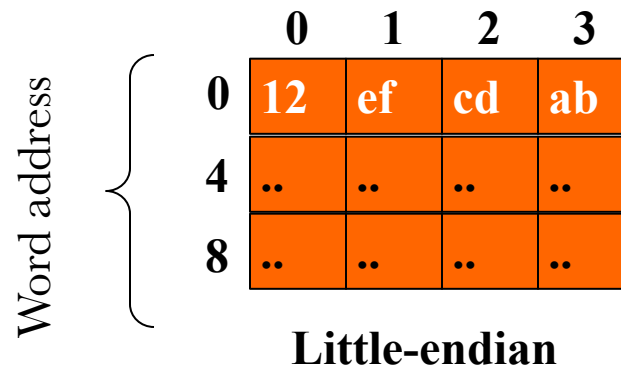
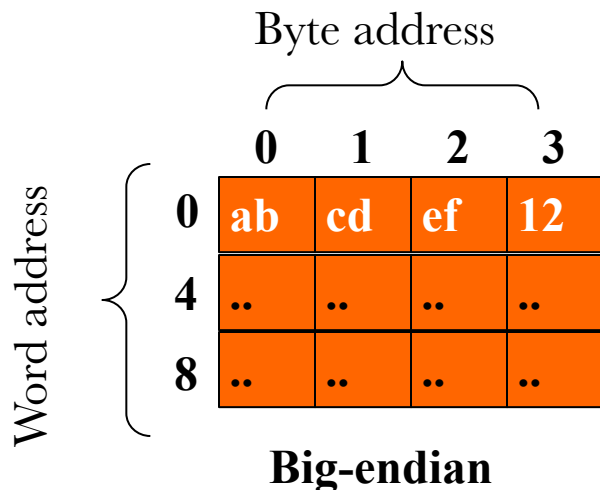
- Bytes are nice, but most data items use larger "words". Register length is 32 bit after all.
- For MIPS (like many other microprocessors), a word is 32 bits or 4 bytes.



- “Alignment restriction” → faster data transfer

# Memory Organization

- $2^{32}$  bytes with byte addresses from 0 to  $2^{32}-1$
- $2^{30}$  words with byte addresses 0, 4, 8, ...  $2^{32}-4$
- Words are aligned
  - words always start at addresses that are multiple of 4.
  - what are the least 2 significant bits of a word address?
- Endian issue: how is  $(0xabcdef12)$  represented in big- and little-endian machines?



# Little or Big-Endian?

```
#define LITTLE_ENDIAN 0  
#define BIG_ENDIAN 1
```

```
int machineEndianness()  
{
```

```
    int i = 1;
```

```
    char *p = (char *) &i;
```

```
    if (p[0] == 1)    // Lowest address contains the least  
                     // significant byte
```

```
        return LITTLE_ENDIAN;
```

```
    else return BIG_ENDIAN;
```

```
}
```

Growing a little endian value in size means adding a byte at the end, whereas with big endian you must move all the bytes one to the right and add the new byte as the beginning

# An Error-Prone Code Segment

...

```
unsigned int x = 0xabcdef01;
```

```
unsigned char *byte_ptr_x;
```

```
byte_ptr_x = (unsigned char *) (&x);
```

```
byte_ptr_x[0] *= 2;
```

```
byte_ptr_x[1] *= 3;
```

```
byte_ptr_x[2] *= 4;
```

```
byte_ptr_x[3] *= 5;
```

...

# Data Transfer Instructions

- Load: moves the data from memory to register
  - **lw** is MIPS instruction for loading a word
- Store: moves the data from register to memory
  - **sw** is MIPS instruction for storing a word
- Example: operands are words.

C code:      `A[12] = h + A[8];`      `/* A is a  
word array */`

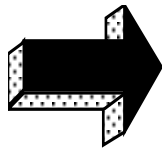
MIPS code: `lw    $t0, 32($s3)    # $s3 = &A[0]  
             add $t0, $s2, $t0   # $s2 = h  
             sw    $t0, 48($s3)`

# Our First Example

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

C code

$\$s4 \rightarrow \&v[]$   
 $\$s5 \rightarrow k$



swap:

```
add    $t2, $s5,    $s5 # $t2=2k  
add    $t2, $t2,    $t2 # $t2=4k  
add    $t2, $s4,    $t2 # $t2=&v[k]  
lw     $t3, 0($t2)      # $t3=v[k]  
lw     $t4, 4($t2)      # $t4=v[k+1]  
sw     $t4, 0($t2)  
sw     $t3, 4($t2)  
jr     $ra
```

# So Far We've Learned:

- MIPS
  - loading words but addressing bytes
  - arithmetic/logic on registers only
- $\$s0, \$s1, \dots$  and  $\$t0, \$t1, \dots$  :  
registers, fast locations for data

- Instruction

Meaning

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

$\$s1 = \$s2 + \$s3$

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

$\$s1 = \$s2 - \$s3$

lw  $\$s1, 400(\$s2)$

$\$s1 = \text{Memory}[\$s2+400]$

sw  $\$s1, 400(\$s2)$

$\text{Memory}[\$s2+400] = \$s1$



# Representing Instructions

- Instructions, like words of data, are represented also as 32-bit numbers
  - Example: `add $t0, $s1, $s2`
- Instruction Format:

0	17	18	8	0	32
---	----	----	---	---	----

Decimal  
representation

000000	10001	10010	01000	00000	100000
--------	-------	-------	-------	-------	--------

6 bits

5 bits

5 bits

5 bits

5 bits

6 bits

Binary  
representation

# Register Numbering

- `$s0` - `$s7` map to 16-23 (for local variables)
- `$t0` - `$t7` map to 8-15 (for temporary variables)
- We will describe the rest later

register	number
<code>\$t0</code>	8
<code>\$t1</code>	9
<code>\$t2</code>	10
<code>\$t3</code>	11
<code>\$t4</code>	12
<code>\$t5</code>	13
<code>\$t6</code>	14
<code>\$t7</code>	15

register	number
<code>\$s0</code>	16
<code>\$s1</code>	17
<code>\$s2</code>	18
<code>\$s3</code>	19
<code>\$s4</code>	20
<code>\$s5</code>	21
<code>\$s6</code>	22
<code>\$s7</code>	23

# Instruction format

<b>op</b>	<b>rs</b>	<b>rt</b>	<b>rd</b>	<b>shamt</b>	<b>funct</b>
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

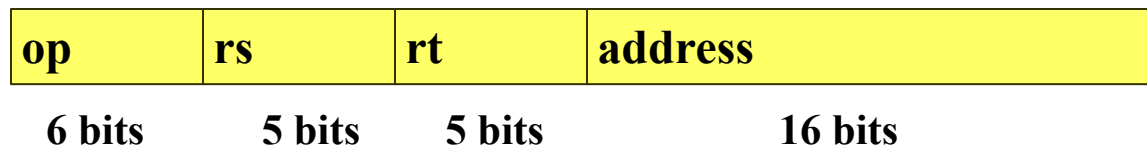
- **op:** *opcode* (type of the operation)
- **funct:** selects the specific variant of the operation in the op field. *Function* but sometimes called *function field*.
- **rs:** The 1<sup>st</sup> register source operand
- **rt:** The 2<sup>nd</sup> register source operand
- **rd:** The 3<sup>rd</sup> register destination operand
- **shamt:** shift amount (explained later)

# Design Principle 3

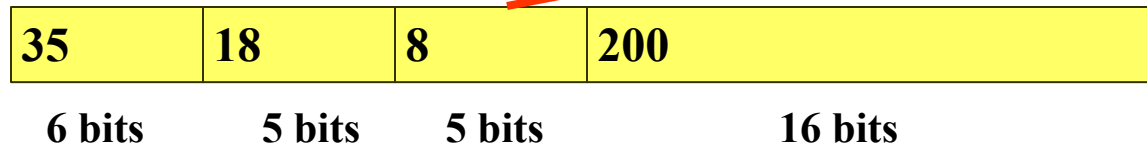
- A potential problem with the previous instruction format
- Consider the “lw” instruction
  - lw must specify two registers and a constant which is the index to the array element.
  - What if this index is in the 5-bit field in the previous format?
  - the first instruction format imposes serious restrictions on data transfer instructions.
- We can have either longer instruction format or second type of instruction format for load.
- Design Principle 3: *Good design demands good compromises.*

# I-type Instruction Format

- MIPS designers elect to introduce a new type of instruction format by keeping the length same:
  - I-type for data transfer instructions
  - other format was R-type for register



- Example: `lw $t0, 200($s2)`



**Notice the 2<sup>nd</sup>  
source register  
becomes the target  
register**

# Constants (Immediate Operands)

- Small constants are used quite frequently
- for example,  $A = A + 5;$   
 $B = B + 1;$   
 $C = D \text{ AND } 0\text{xff};$
- The constants can be loaded from the memory, or
- The constants can be kept inside the instruction.
- Example: `addi $sp, $sp, 4 # add immediate`  
The opcode of `addi` is 8 and the instruction format

I type

op	rs	rt	16-bit address
001000	11101	11101	0000 0000 0000 0100

# Design Principle 4

- Design Principle 4: *Make the common case fast.*
- Constant operands occur frequently, and by making constants part of arithmetic instructions, they are much faster than if they were loaded from memory.

# The Instructions We've Learned

Instruction	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32	n.a.
sub	R	0	reg	reg	reg	0	34	n.a.
lw	I	35	reg	reg	n.a.	n.a.	n.a.	address
sw	I	43	reg	reg	n.a.	n.a.	n.a.	address
addi	I	8	reg	reg	n.a.	n.a.	n.a.	constant

- Example:  $A[300] = h + A[300]$

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

**Machine code**

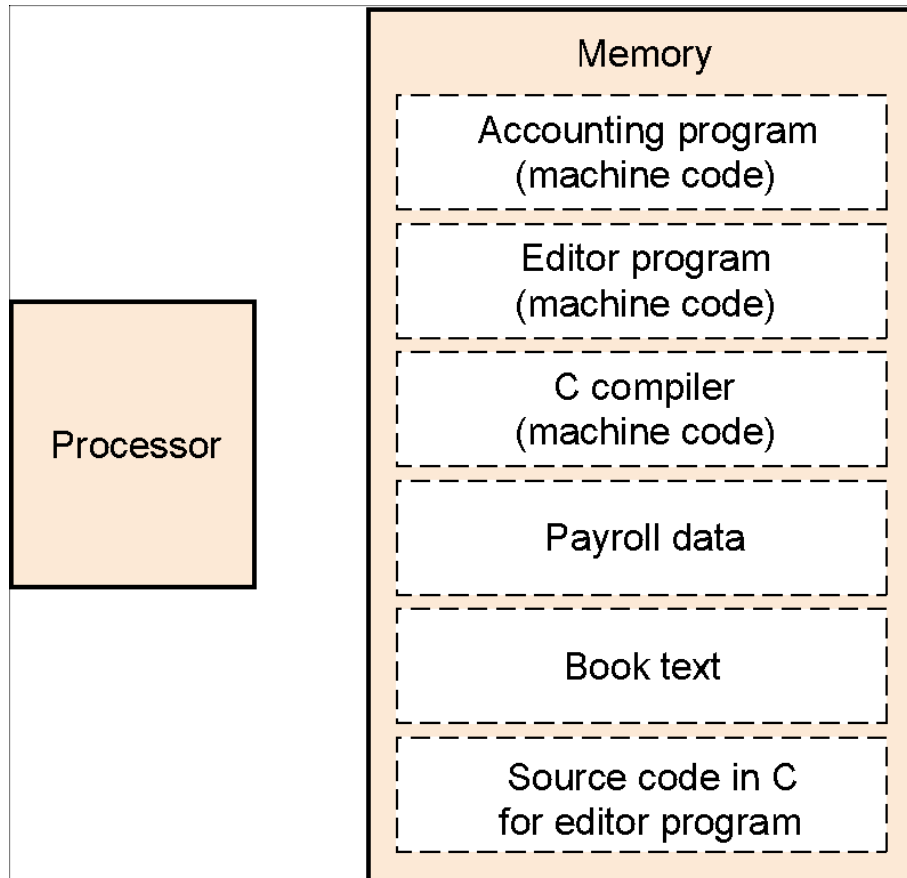
```
lw    $t0, 1200($t1)
add   $t0, $s2, $t0
sw    $t0, 1200($t1)
```

**Assembly code**



# Stored-Program Concept

- Programs are stored in memory (to be read or written just like data)



- Fetch & Execute Cycle
  1. Instructions are fetched and put into a special register
  2. Bits in the register "control" the subsequent actions
  3. Fetch the "next" instruction and continue

# Instruction for Control Flow

- Decision making instructions
  - alter the control flow,
  - i.e., change the "next" instruction to be executed

- MIPS *conditional branch* instructions:

`bne $t0, $t1, Label`

`beq $t0, $t1, Label`

- Example:     `if (i == j)`  
                  `h = i + j;`

`bne $s0, $s1, Label`

`add $s3, $s0, $s1`

`Label: ...`

# Jump Instruction

- MIPS unconditional branch instructions:

j label

- Example:

```
if (i != j)          beq $s4, $s5, Lab1
    h = i+j;         add $s3, $s4, $s5
else                 j Lab2
    h = i-j;         Lab1: sub $s3, $s4, $s5
                    Lab2: ...
```

J-type instruction for unconditional jump

op	address
6 bits	26 bits

# Loop

- Example:

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

$i$ ,  $j$ ,  $k$  corresponds to  $\$s3$ ,  $\$s4$ ,  $\$s5$ , respectively and base address of the array in  $\$s6$

- Loop:

```
add $t1, $s3, $s3    # $t1 = 2i
add $t1, $t1, $t1    # $t1 = 4i
add $t1, $t1, $s6    # $t1 = &save[i]
lw  $t0, 0($t1)      # $t0 = save[i]
bne $t0, $s5, Exit   # go to Exit
                        # if save[i] ≠ k
add $s3, $s3, $s4    # i = i + j
j    Loop            # go to Loop
```

Exit:

# So Far

- Instruction

## Meaning

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

$\$s1 = \$s2 + \$s3$

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

$\$s1 = \$s2 - \$s3$

lw \$s1, 400(\$s2)

$\$s1 = \text{Memory}[\$s2 + 400]$

sw \$s1, 400(\$s2)

$\text{Memory}[\$s2 + 400] = \$s1$

bne \$s4, \$s5, Label

Next instr. @ Label if

$\$s4 \neq \$s5$

beq \$s4, \$s5, Label

Next instr. @ Label if

$\$s4 = \$s5$

j Label

Next instr. @ Label

- Formats:

R type

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

I type

op	rs	rt	16-bit address
----	----	----	----------------

J type

op	26-bit address
----	----------------

# More Control Flow

- We have: beq and bne
  - what about *Branch-if-less-than* ?
- New instruction: set on less than  
`slt $t0, $s1, $s2    # $t0 = 1 if $s1 < $s2  
                              # 0             otherwise`
- slt is used to build blt \$s1, \$s2, Label
- What about bge \$s1, \$s2, Label
- MIPS doesn't include blt or bge since they either require a slower clock or takes two clock cycles to execute.

# Policy of Register Use Conventions

Name	Register no	Usage	Preserved on call
\$zero	0	the constant value 0	n.a.
\$at	1	Assembler temporary	no
\$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

Registers 26-27, called \$k0-\$k1, are reserved for OS.

# Procedure

- A tool used to structure programs into small segments.
- Reusing code segments
- Steps
  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 results in a place where the calling program can access it.
  6. Return control to the point of origin



# Register Assignments

- `$a0–$a3`: Four argument registers
- `$v0–$v1`: two value registers
- `$ra`: return register to return to the point of origin.
- The instruction is jump-and-link (`jal`)  
`jal ProcedureName`
- The register PC (*program counter*) is used to hold the address of the current instruction.

# Procedure Call

- `jal` saves `PC+4` in register `$ra` for the procedure return.
  - `jr $ra` # jump register
- Caller puts the arguments in `$a0` – `$a3` and uses `jal A` to jump to procedure `A`.
- Callee performs the task, places the results in `$v0`–`$v1`, and executes `jr $ra`
- To store additional variables, stack is used.

# Stack

- Stack is a LIFO queue.
- A stack pointer ( $\$sp$ ) contains the address of the most recently allocated address (*top of the stack*).
- Stack has two operations: “push” and “pop”.
- Stacks, in MIPS, grow from higher to lower addresses.
  - “push” decrement  $\$sp$ .
  - “pop” increment  $\$sp$ .

# Example 1/2

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

• g, h, i, j → \$a0, \$a1, \$a2, \$a3 and f → \$s0

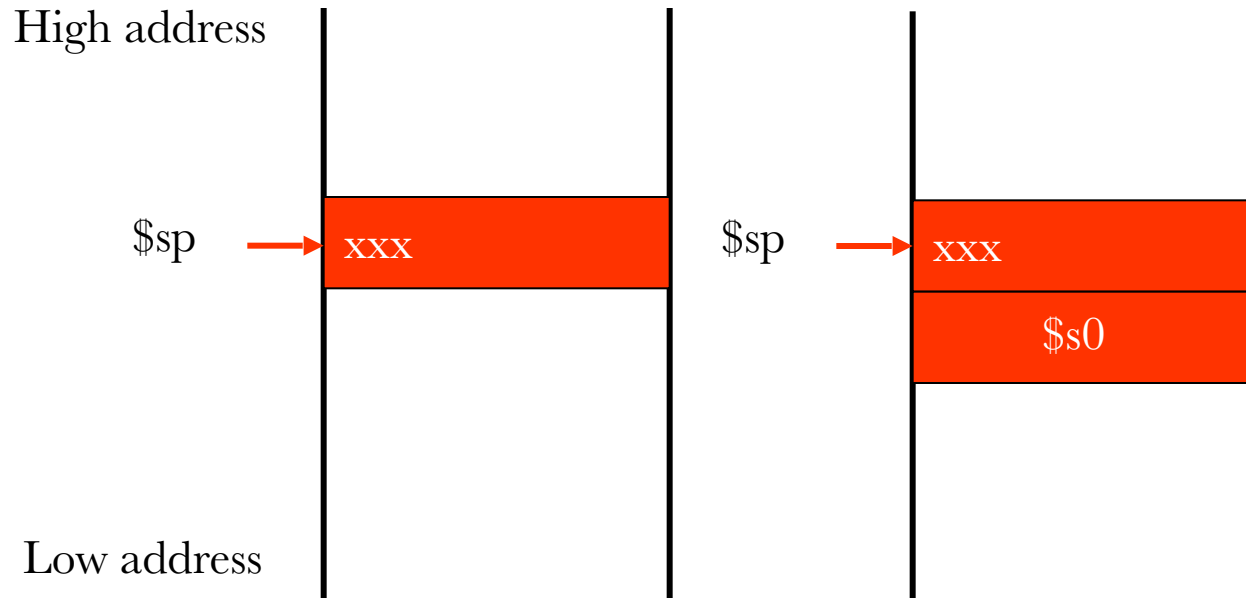
leaf\_example:

```
addi $sp, $sp, -4    # make room for one item
sw   $s0, 0($sp)     # save register $s0
```

...

• Content of \$s0 is saved on stack since the callee is going to use it.

# Stack Activity



# Example 2/2

...

# \$s0 saved

```
add $t0, $a0, $a1      # $t0 = g+h
add $t1, $a2, $a3      # $t1 = i+j
sub $s0, $t0, $t1      # $s0 = $t0 - $t1
add $v0, $s0, $zero    # $v0 = $s0 + 0 (returns f)
```

...

- before returning, restore the old values of the registers for the caller

...

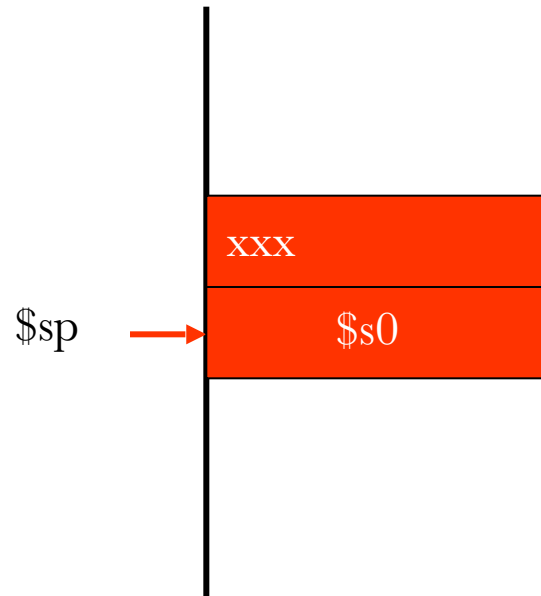
```
lw  $s0, 0($sp)        # restore register $s0
add $sp, $sp, 4        # delete 1 item
```

...

- return the control to the caller

```
jr  $ra                # jump back to the caller
```

# Stack Activity



# Nested Procedures

- main procedure  $\rightarrow$  Procedure A  $\rightarrow$  Procedure B
  - Both main program and Procedure A use the same registers ( $\$a0-\$a3$ ) to pass arguments
  - Both procedure A and procedure B use the same registers ( $\$v0-\$v1$ ) to return results.
  - There is a conflict over the  $\$ra$  register as well.
- Solution:
  1. The caller pushes ( $\$a0-\$a3$ ) and ( $\$t0-\$t9$ ) that are needed after the call
  2. The callee pushes the return address ( $\$ra$ ) and any unsaved registers ( $\$s0-\$s7$ ) that it plans to use. It also promises to adjust  $\$sp$ .



# Nested Procedures: Example 1

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

- Assembly code:

```
fact:
    addi $sp, $sp, -8      # make room for two items
    sw   $ra, 4($sp)      # save the return address
    sw   $a0, 0($sp)      # save the argument n

    slti $t0, $a0, 1      # test if n < 1
    beq  $t0, $zero, L1    # if n >= 1 then go to L1
    ...
```

# Nested Procedures: Example 2

...

```
addi $v0, $zero, 1    # return 1
addi $sp, $sp, 8      # pop 2 items off the stack
jr $ra
```

```
L1: subi $a0, $a0, 1    # n >= 1, put n-1 in $a0
     jal fact           # call fact with n-1
```

- where fact returns (result is available in \$v0)

```
lw $a0, 0($sp)         # return from jal, restore n
lw $ra, 4($sp)          # restore the return address
addi $sp, $sp, 8        # pop two items
mul $v0, $a0, $v0       # return n * fact(n-1)
jr $ra                  # return to the caller
```

# What is Preserved Across the Call?

Preserved	Not Preserved
Saved registers: $\$s0-\$s7$	Temporary registers: $\$t0-\$t9$
Stack Pointer: $\$sp$	Argument registers: $\$a0-\$a3$
Return address: $\$ra$	Return value registers: $\$v0-\$v1$
Stack above the stack pointer	Stack below the stack pointer

- The caller saves
  - $\$a0-\$a3$  and  $\$t0-\$t9$ , (possibly  $\$v0-\$v1$ )
- The callee
  - saves  $\$ra$  and ( $\$s0-\$s7$ ) that it uses.
  - promises not to write the portion of the stack above  $\$sp$  and restore the old value of  $\$sp$  when it returns

# MIPS Memory Allocation

sp → 0x7fff fffc

Stack



Dynamic Data



gp → 0x1000 8000

0x1000 0000

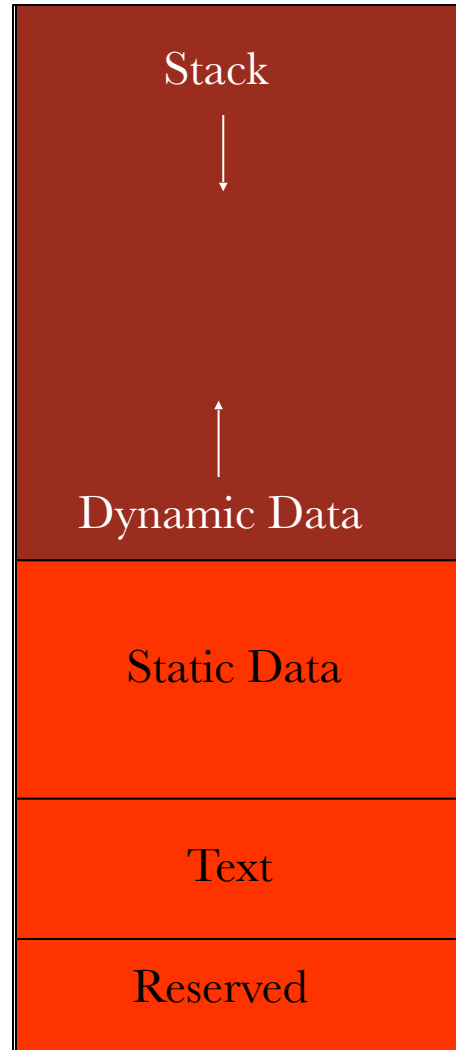
Static Data

PC → 0x0040 0000

Text

Reserved

0



# 32-bit Immediate Operands

- Constants can sometimes get longer than 16-bits.

`$t0 = 4,042,325,760 =`

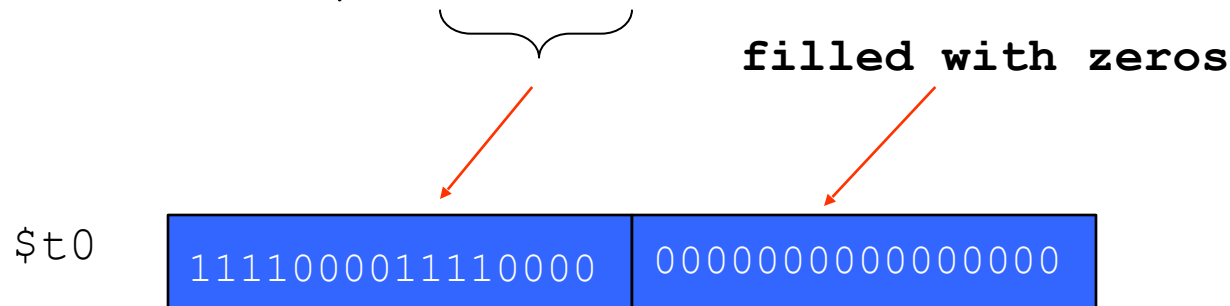
`11110000111100001111111100000000`

- There is a special instruction load upper immediate

```
lui $t0, 61680 # $t0 is register 8
ori $t0, $t0, 65280
```

# Long Constants

- `lui $t0, 61680`



- Then it must get the lower order bits right, i.e.,

`ori $t0, $t0, 65280`



**OR**



can we do it with `addi`?



# Assembly vs. Machine Language

- Assembly provides convenient symbolic representation
- Machine language is the underlying reality
- Assembly can provide 'pseudo-instructions' (i.e. instruction that do not exist in machine language) for convenience
  - e.g., “`move $t0, $t1`” exists only in Assembly
  - would be implemented using “`add $t0, $t1, $zero`”
  - `blt $at, $s0, $s1`
- When considering performance you should count the real instructions

# Addresses in Branches and Jumps

j Label	#Next instruction is at Label
bne \$t4, \$t5, Label	#Next instruction is at Label #if \$t4 $\neq$ \$t5
beq \$t4, \$t5, Label	# Next instruction is at Label # if \$t4 = \$t5

I type

op	rs	rt	16-bit address
----	----	----	----------------

J type

op	26-bit address
----	----------------

- In bne and beq, the target address cannot be bigger than  $2^{16}-1$ .
- Our programs cannot have more than  $2^{16}$  instructions.
- We use 32-bit addresses for memory where the instructions are.



# Addresses in Branches

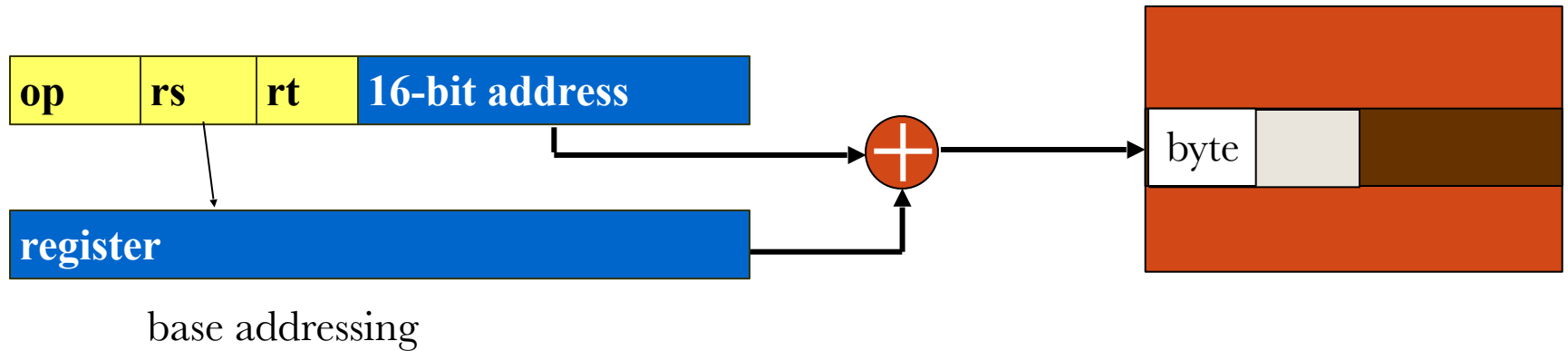
- Remedy: PC-relative addressing
- `branch address = PC + 16-bit address`
- Most branches are local (*principle of locality*).
- Procedures have no reason to be near the call.
  - Therefore, MIPS architecture offers long addresses for procedure calls by using the J-type format for both `jump` and `jump-and-link` instructions.
  - Uses pseudo-direct addressing

# MIPS Addressing Modes 1/2

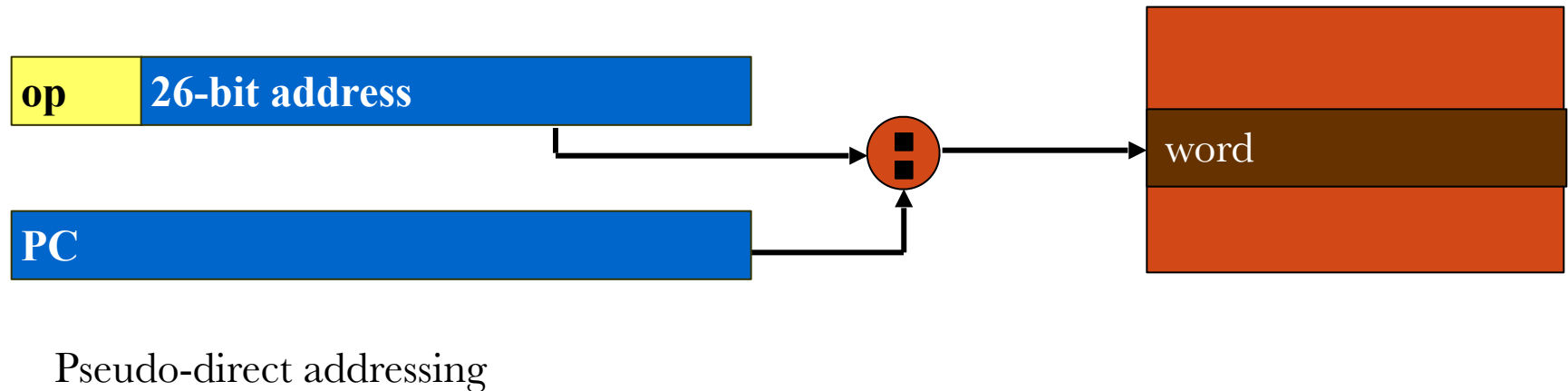
- Immediate addressing,
  - operand is a constant within the instruction
- Register addressing,
  - operands are in registers
- Base or displacement addressing,
  - operand is at the memory location whose address is the sum of a register and a constant in the instruction
- PC-relative addressing,
  - address is the sum of PC and a constant in instruction
- Pseudo-direct addressing,
  - the jump address is the 26 bits of the instruction concatenated with the upper bits of the PC

# MIPS Addressing Modes 2/2

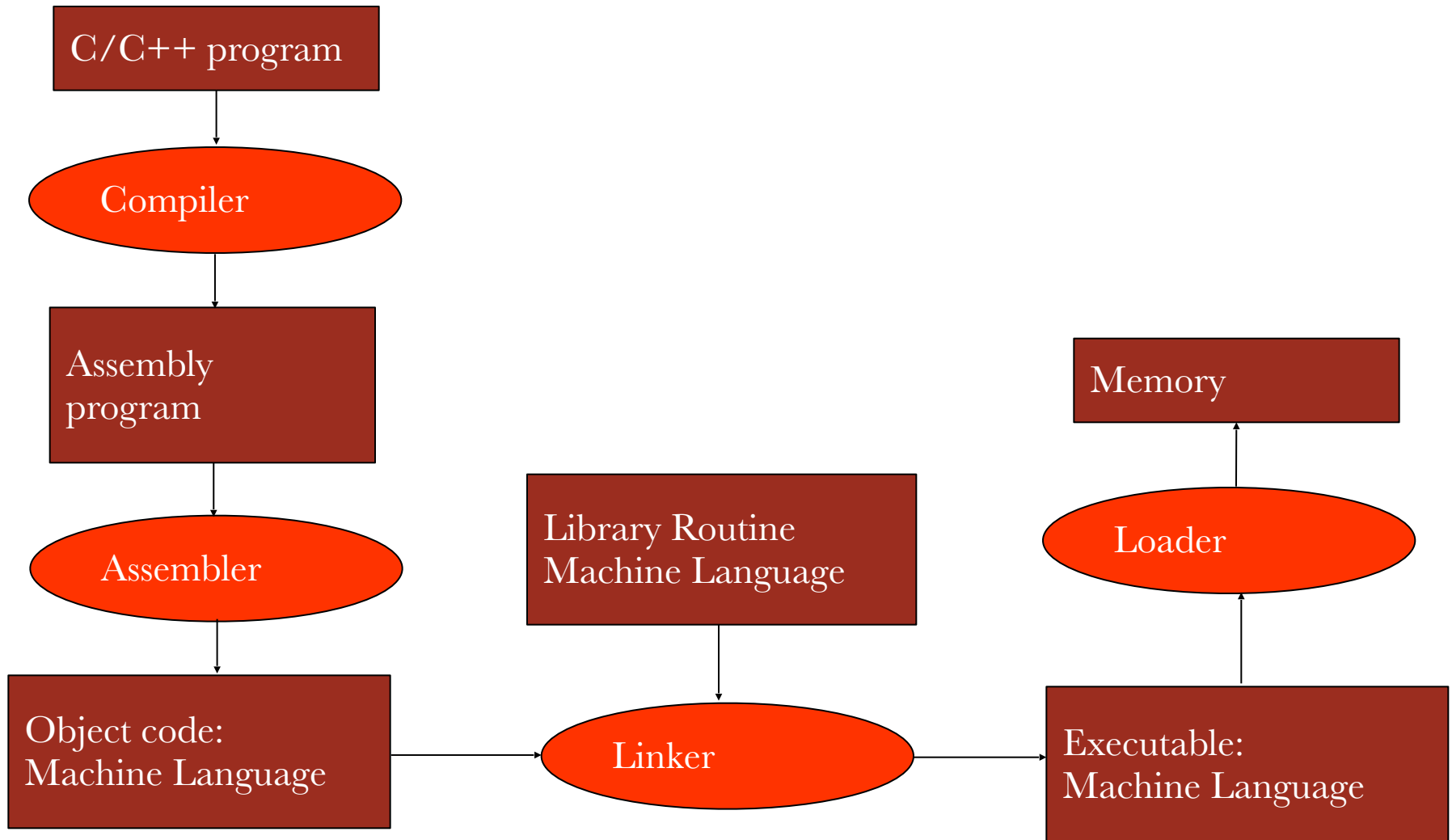
```
lb $t2, index($t1)
```



```
jal ProcedureAddress
```



# Starting a Program



# Assembler 1/2

- Assembler turns the assembly program into an object file
  - Object file = machine language instructions + data + information needed to place instructions properly in memory.
- Six pieces of object file (for Unix):
  1. Object file header: describes the size and position of the other pieces in the object file.
  2. Text segment: contains the machine language.
  3. Data segment: contains the data that comes with the program.  
Static and dynamic data.

# Assembler 2/2

- Six pieces of object file (for Unix):
  4. Relocation information: identifies the instructions and data words that depend on absolute addresses when the program is loaded into memory.
  5. Symbol table: contains the remaining labels that are not defined, such as external references.
  6. Debugging information: contains a concise description of how the modules were compiled so that a debugger can associate machine instructions with C source files and make data structures readable.

# Linker 1/2

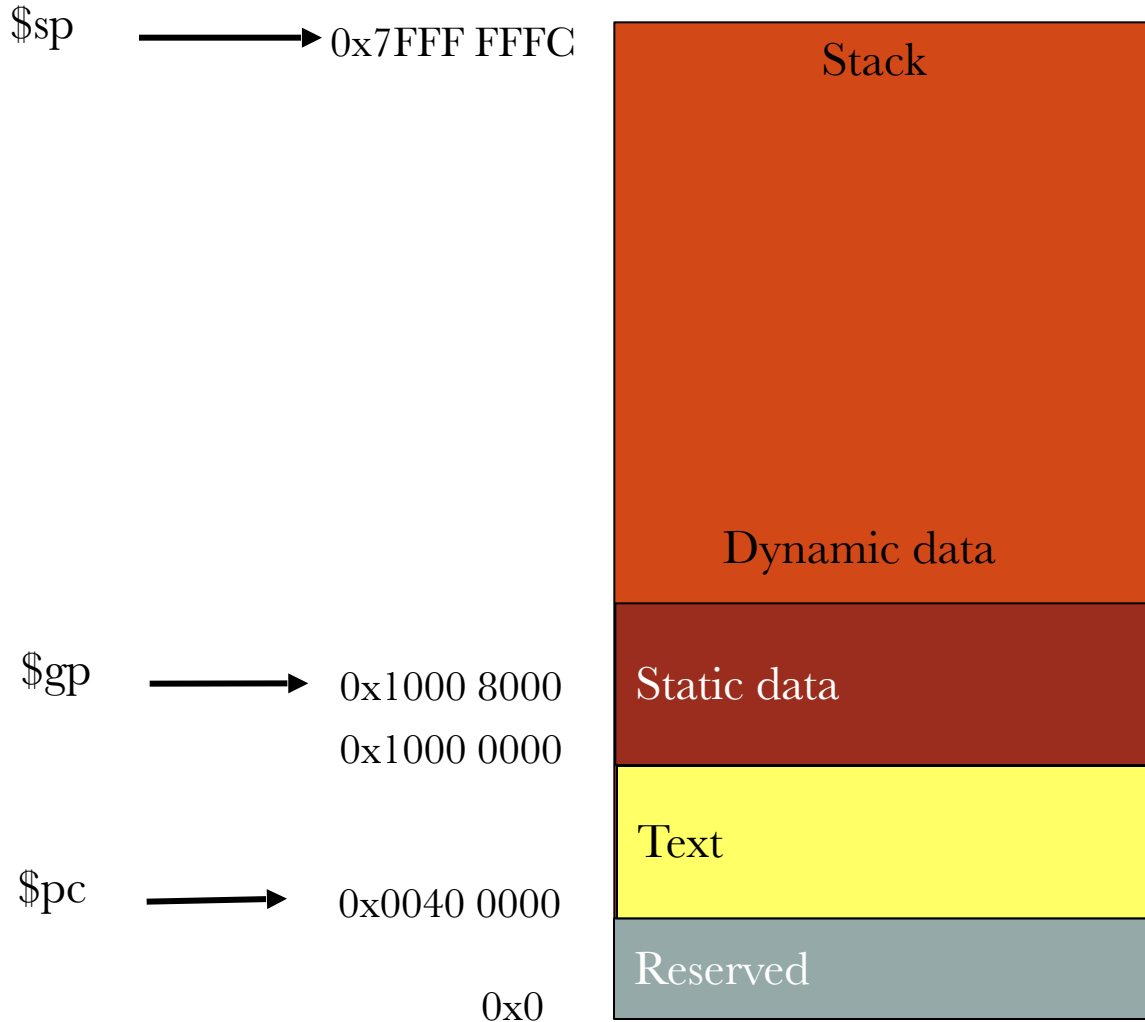
- General approach:
  - compile and assemble each procedure independently
  - link them by link editor or linker.
- Three link steps:
  1. Place code and data modules symbolically in memory
  2. Determine the addresses of data and instruction labels.
  3. Patch both internal and external references.
- The linker uses the relocation information and symbol table in each object module to resolve all undefined labels.

# Linker 2/2

- Unresolved references can occur in branch and jump (`jb` and `jnal`) instructions, data addresses
  - Linker finds those references and replace them with absolute addresses.
  - Linker produces an executable file that can run on computer
  - The format is the same as object file but it contains no unresolved references, relocation information, symbol table, (or debugging information).



# MIPS Memory Allocation



# Example 1/3

<b>Object file header</b>	name	procedure A	
	text size	0x0100	
	data size	0x0020	
<b>Text segment</b>	address	instruction	
	0	lw \$a0, 0(\$gp)	lw \$a0, X
	4	jal 0	jal B
	...	...	
<b>Data segment</b>	0	(X)	
	...	...	
<b>Relocation information</b>	address	instruction type	Dependency
	0	lw	X
	4	jal	B
<b>Symbol Table</b>	Label	Address	
	X	-	
	B	-	

# Example 2/3

<b>Object file header</b>	Name	Procedure B	
	text size	0x0200	
	data size	0x0030	
<b>Text segment</b>	address	instruction	
	0	lw \$t0, 0(\$gp)	lw \$t0, Y
	4	j 0	j Loop
	...	...	
	16	Loop: addi \$t0, \$t0, 1	
	...	...	
<b>Data segment</b>	0	(Y)	
	...	...	
<b>Relocation information</b>	address	instruction type	Dependency
	0	lw	Y
	4	j	Loop
<b>Symbol Table</b>	Label	Address	
	Y	-	
	Loop	-	

# Example 3/3

<b>Executable file header</b>	text size	0x0300
	data size	0x0050
<b>Text segment</b>	address	instruction
	0x0040 0000	lw \$a0, 8000(\$gp)
	0x0040 0004	jal 0040 0100
	...	...
	0x0040 0100	lw \$t0, 8020(\$gp)
	0x0040 0104	j 0040 0110
	...	...
	0x0040 0110	Loop: addi \$t0, \$t0, 1
	...	...
	...	...
<b>Data segment</b>	Address	
	0x1000 0000	(X)
	...	...
	0x1000 0020	(Y)
	...	...

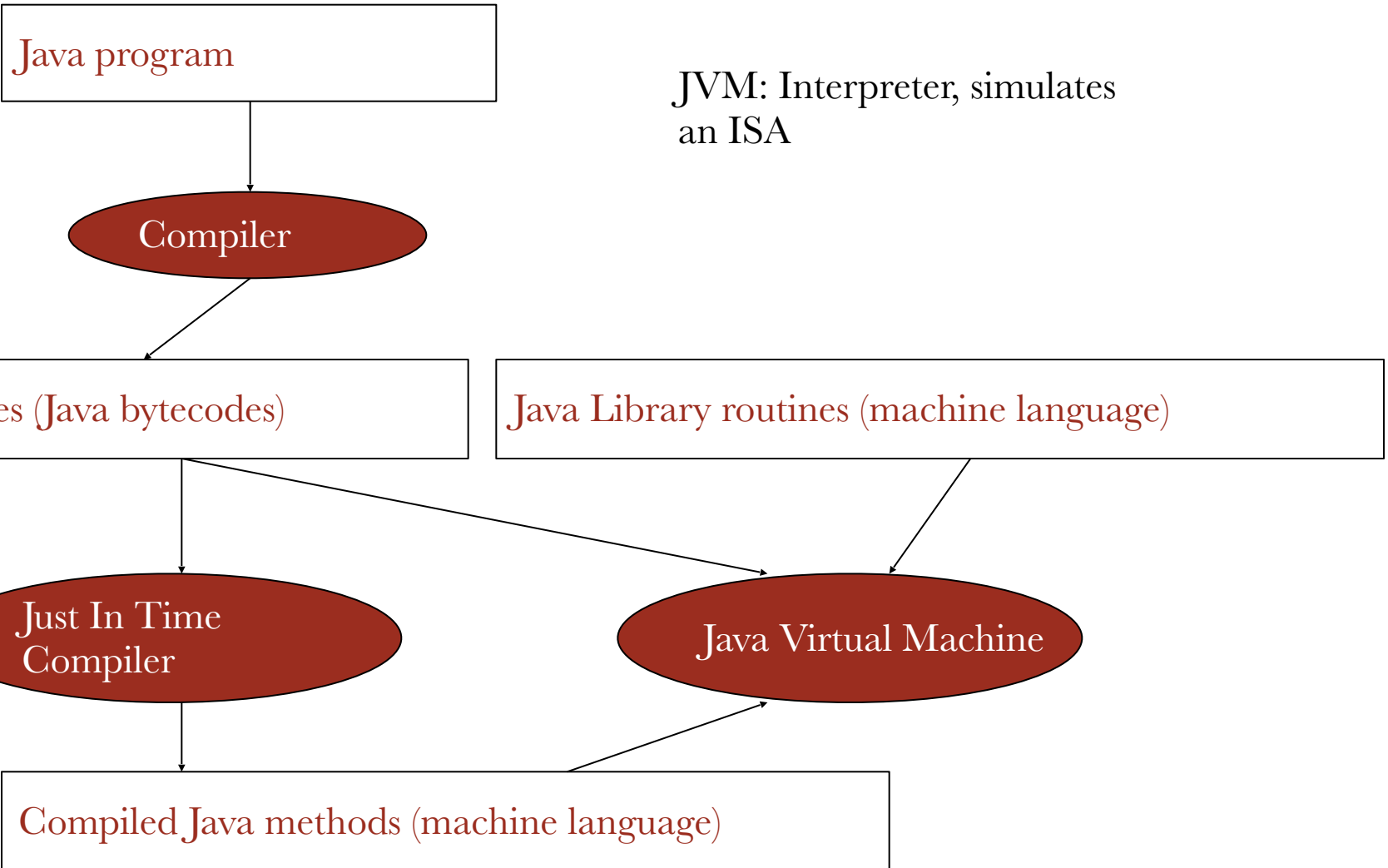
# Loader

- Steps

1. read the executable file header to determine the size of text and data segments
2. create an address space large enough for the text and data
3. copy the instructions and data from the executable file into memory
4. copy the parameters (if any) to the main program onto the stack
5. initialize the machine registers and set the stack pointer to the first free location
6. jump to a “start-up” routine that copies the parameters into the argument registers and calls the main routine. When the main routine returns, the start-up routine calls `exit` (a system call).

# Starting a Java Program

JVM: Interpreter, simulates  
an ISA



# Intel Processors 1/3

- Intel 8086 (1978): 16 bit architecture,
  - 29K Transistors, 20-bit address
  - The floating point coprocessor (8087) is added (1980)
- 80286 (1982): Address space is 24 bits,
  - instructions added
  - 134 K transistors, the basis of IBM PC-AT personal computer
- 80386 (1985): 32 bit architecture,
  - 275 K transistors
  - Flat addressing model, first Intel architecture that could support Unix OS.
- i486 (1989):
  - Improved performance. Integration of floating-point unit onto the processor-chip.
  - Instruction set is the same, 1.9M transistors

# Intel Processors 2/3

- Pentium (1992): Improved performance,
  - 3.1 M transistors
  - minor additions to the instruction set
- PentiumPro (1995): a new design (internally P6),
  - conditional move instructions, L2 cache on package
  - 6.5 M transistors
- Pentium/MMX (1997): multimedia and communications applications
  - 4.5M transistors
  - New class of instructions for manipulating vector of integers. SIMD (3DNow! Architecture by AMD)
- Pentium II (1997): Merge P6 with MMX
  - 7 M transistors.
  - MMX instructions are implemented in P6 architecture



# Intel Processors 3/3

- Pentium III (1999):
  - 8.2 M transistors (later versions up to 24 M transistors)
  - another class of instructions for manipulating vectors of integer
  - SSE (Streaming SIMD Extensions)
  - Eight separate 128 bit (MMX) registers (XMM0–XMM7)
  - MMX and floating point instructions execute at the same time
- Pentium 4 (2001):
  - 42 M transistors
  - 144 new instructions (SSE2)
  - Two 64-bit floating point operations can execute in parallel
- AMD64 (EM64T) (2003-2004):
  - 64-bit address space
  - 64-bit registers
  - *long mode*
  - *legacy mode and compatibility mode*

# A Dominant Architecture: 80386

- 16 Registers (8 of them are general purpose)
- Instructions “from 1 to 17 bytes long
- complex addressing modes
  - e.g., “base or scaled index with 8 or 32 bit displacement”
  - $\text{address} = \text{base} + (2^{\text{scale}} \times \text{Index}) + \text{displacement}$
- Conditional branches: condition codes (a register) are set as a side effect of an operation

# Arithmetic/Logic Instructions

Source/Destination operand	Second Source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

# 80386 Register Set

Name	31	0	Use
EAX			GPR 0
ECX			GPR 1
EDX			GPR 2
EBX			GPR 3
ESP			GPR 4
EBP			GPR 5
ESI			GPR 6
EDI			GPR 7
	CS		Code segment pointer
	SS		Stack segment pointer (top of stack)
	DS		Data segment pointer 0
	ES		Data segment pointer 1
	FS		Data segment pointer 2
	GS		Data segment pointer 3
EIP			Instruction pointer (PC)
EFLAGS			Condition codes

# Condition Codes: EFLAGS

- CF: Carry Flag
  - The most recent operation generated a carry out.
- ZF: Zero Flag
  - The most recent operation yielded zero.
- SF: Sign Flag
  - The most recent operation yielded a negative value
- OF: Overflow Flag
  - The most recent operation caused a two's complement overflow – either positive or negative

# Typical 80386 Instruction Formats



JS name

if negative (condition code)  
go to [EIP + displacement]



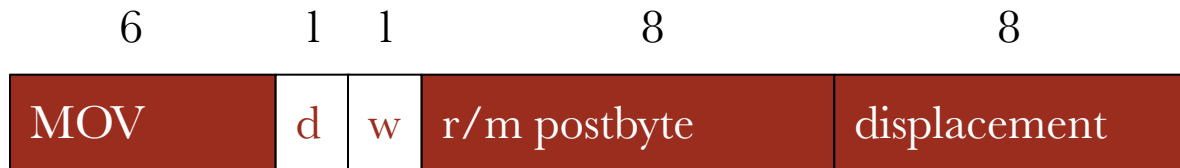
CALL name

$SP = SP - 4$

$M[SP] = EIP + 5$

$EIP = Offset$

PC



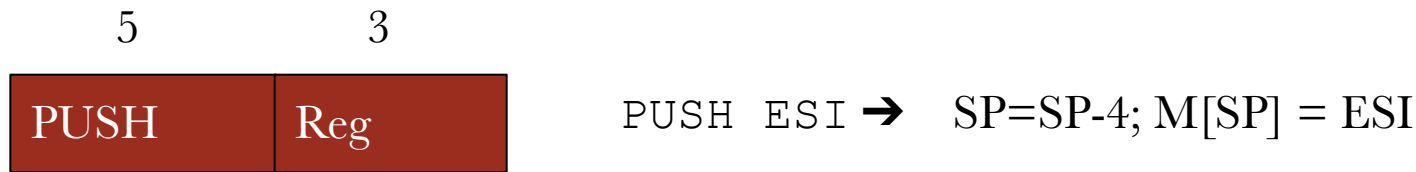
direction

byte or  
word?

specify target register,  
which registers are used  
in address calculation, how?

MOV EBX, [EDI+45]

# Typical 80x86 Instruction Formats



TEST EDX, #42 → set condition code flags with EDX = 0x42

# Summary

- Instruction complexity is only one variable
  - lower instruction count vs. higher CPI / lower clock rate
- Design Principles:
  - simplicity favors regularity
  - smaller is faster
  - good design demands good compromise
  - make the common case fast
- Instruction set architecture
  - a very important abstraction indeed!