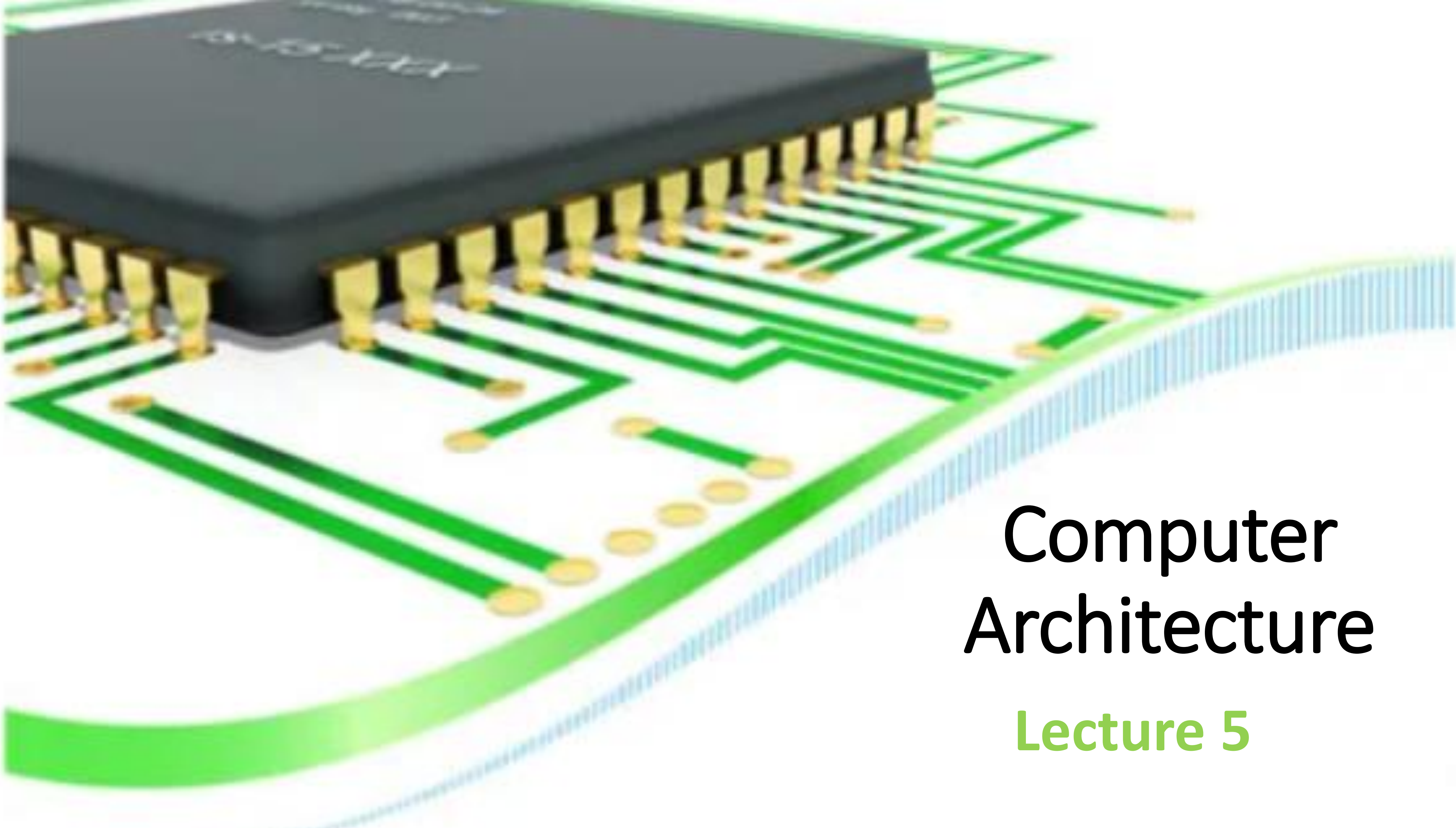
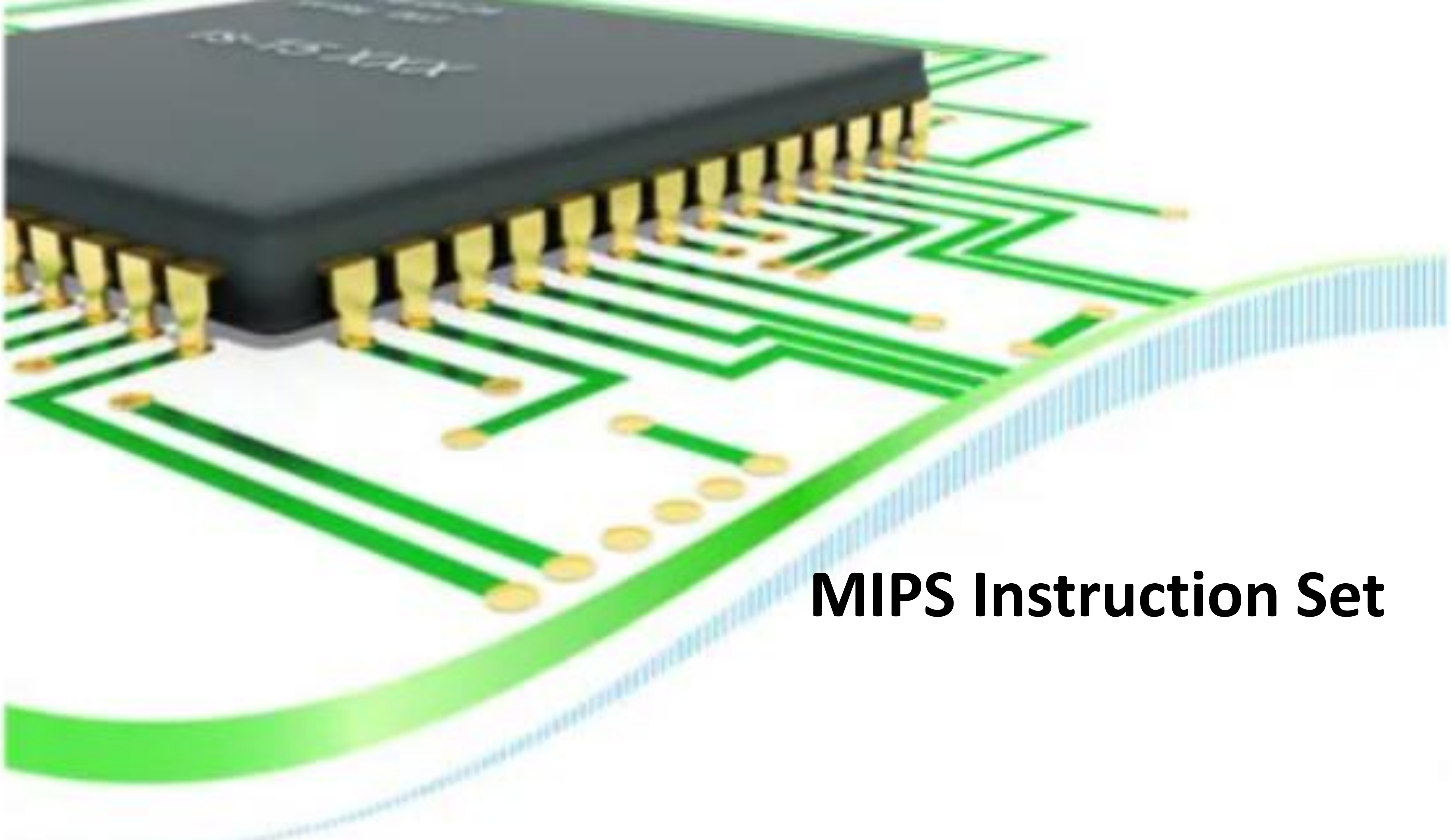


Computer Architecture

Lecture 5

Agenda

- MIPS Instruction Set Architecture
- Writing programs in MIPS
- MIPS instruction formats
- Translating and starting a program



MIPS Instruction Set

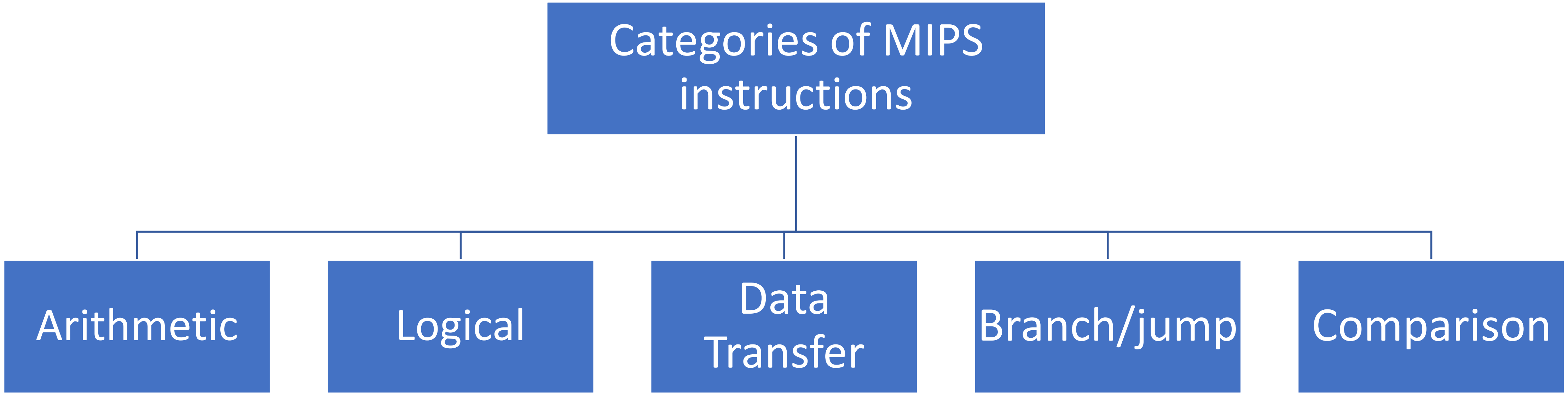
MIPS-32 ISA

- MIPS-32 uses 32 general purpose registers, each 32 bits wide
- The MIPS architecture can support up to 32 address lines.
- MIPS is a byte-addressable architecture
- The word size is 32-bits

MIPS commonly used registers

Register Number	Register Name	Description
0	$zero	The value 0
2-3	$v0 - $v1	(values) from expression evaluation and function results
4-7	$a0 - $a3	(arguments) First four parameters for subroutine
8-15, 24-25	$t0 - $t9	Temporary variables
16-23	$s0 - $s7	Saved values representing final computed results
31	$ra	Return address

Categories of MIPS instructions



Arithmetic instructions

add \$s2, \$s1, \$s0

destination register

Source operand registers

\$s0	\$s1	\$s2
5	3	8

sub \$s2, \$s0, \$s1 # \$s2 = 2

mul \$s2, \$s1, \$s0 # \$s2 = 15

div \$s2, \$s0, \$s1 # \$s2 = 1

Arithmetic instruction

addi $s2, $s1, 3

Immediate value

destination register

Source operand register

$s1

5

$s2

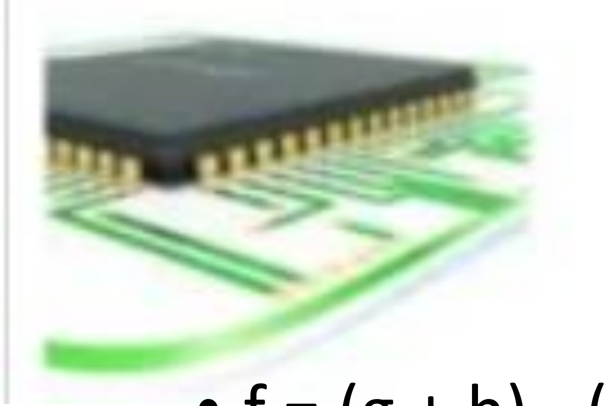
8

li $s4, 50

Data transfer instruction

$s4

50



MIPS Assembly language Example

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

- f in $s0, g in $s1, h in $s2, i in $s3, j in $s4
- we use t registers for storing temporary values
- Compiled MIPS code:

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

MIPS Assembly Example

```
a = 5
b = 3
c = 10
x = a+b
y = x+7
x = y- a
z= c  * x
result = z/a
```

```
li $s0, 5
li $s1, 3
li $s2, 10
add $t0, $s0, $s1
addi $t1, $t0, 7
sub $t0, $t1, $s0
mul $t3, $t0, $s2
div $s3, $t3, $s0
```

```
# Declare main as a global function
.globl main
# All program code is placed after the
# .text assembler directive
.text

# The label 'main' represents the starting point
main:
 li $s0, 5
 li $s1, 3
 li $s2, 10
 add $t0, $s0, $s1
 addi $t1, $t0, 7
 sub $t0, $t1, $s0
 mul $t3, $t0, $s2
 div $s3, $t3, $s0
 li $v0, 10 # Sets $v0 to "10" to select exit syscall
 syscall # Exit
```

R0	[r0]	=	0
R1	[at]	=	0
R2	[v0]	=	10
R3	[v1]	=	0
R4	[a0]	=	7
R5	[a1]	=	2147481548
R6	[a2]	=	2147481580
R7	[a3]	=	0
R8	[t0]	=	10
R9	[t1]	=	15
R10	[t2]	=	0
R11	[t3]	=	100
R12	[t4]	=	0
R13	[t5]	=	0
R14	[t6]	=	0
R15	[t7]	=	0
R16	[s0]	=	5
R17	[s1]	=	3
R18	[s2]	=	10
R19	[s3]	=	20

MIPS Assembly Example

```
a = 6;
b=54
c=0;
if (a>b)
    c = x+y;
else
    c = x-y;
```

Branch if greater than

```
 li $s0, 6
 li $s1, 4
 bgt $s0, $s1, ifpart
 sub $s2, $s0, $s1
 j Exit

ifpart:
 add $s2, $s0, $s1

Exit:
 li $v0, 10
 syscall
```

label

jump

Check your Understanding


```
a = 6;
b=4;
max = 0;
if (a>b)
    max = a;
else
    max = b;
```

```
 li $s0, 6
 li $s1, 4
 bgt $s0, $s1, ifpart
 move $s2, $s1
 j Exit

ifpart:
 move $s2, $s0

Exit:
 li $v0, 10
 syscall
```

MIPS Assembly

Assume that first memory cell to store the program available is address 200 in decimal

Symbol Table

label	Address
ifpart	220
Exit	228

```
200 li $s0, 6
204 li $s1, 4               220
208 bgt $s0, $s1, ifpart
212 sub $s2, $s0, $s1
216 j Exit  228

220 ifpart:
224  add $s2, $s0, $s1

228 Exit:
232  li $v0, 10
236  syscall
```

A label is an address

MIPS Instructions different forms

add $s2, $s1, $s0
sub $s2, $s1, $s0

instructions with three registers

addi $s2, $s1, 10

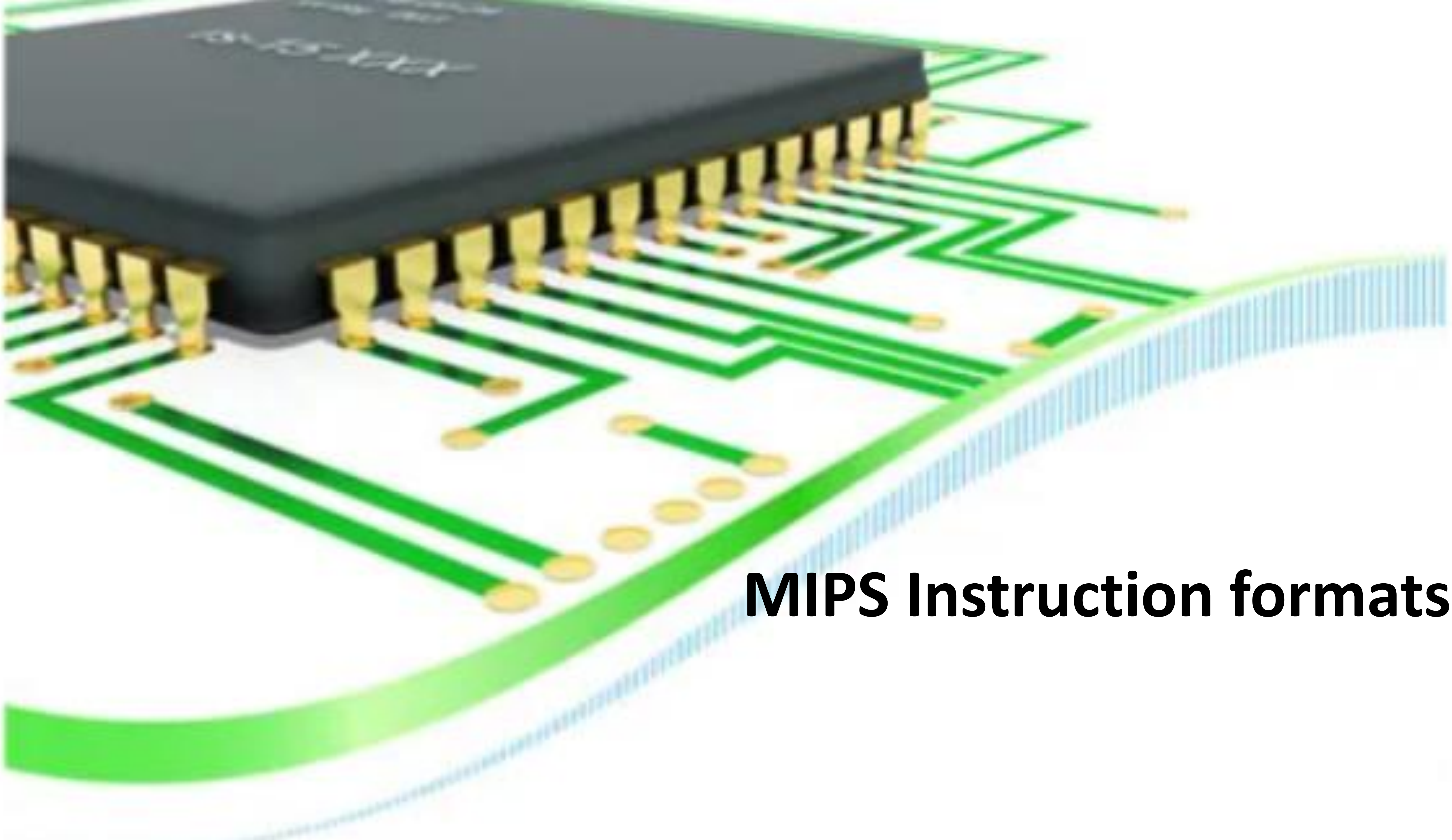
bgt $s2, $s1, 300

instructions with two registers and a value

j 400

instructions with one value

A label is an address



MIPS Instruction formats

Instruction format

- The instruction format defines the layout of the bits for an instruction.

- The instruction formats are a sequence of bits (0 and 1). These bits, when grouped, are known as fields. Each field of the machine provides specific information to the CPU related to the operation and location of the data.

MIPS Instruction Formats

add \$s2, \$s1, \$s0
sub \$s2, \$s1, \$s0

instructions with three registers

R-format

addi \$s2, \$s1, 10

bgt \$s2, \$s1, 300

instructions with two registers and a value

I-format

j 400

instructions with one value

J-format

MIPS Instruction Formats

- Instruction formats in MIPS
  - R-format
  - I-format
  - J-format

R-format instructions

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

32 bits

$s0 → R16
$s1 → R17
$s2 → R18

add $s2, $s0, $s1

Representation in decimal →

0	**16**	**17**	**18**	**0**	**32**

Representation in binary →

000000	**10000**	**10001**	**10010**	**00000**	**100000**

I-format instructions

opcode	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

32 bits

\$s0 → R16
\$s1 → R17

addi \$s1, \$s0, 20

Representation in decimal →

8	**16**	**17**	**20**

Representation in binary →

001000	**10000**	**10001**	**010100**

I-format instructions

opcode	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

32 bits

*N.B.: Branch, some load and store instructions are **I-format** instructions that have exceptions when translating them to binary → we might discuss that later*

J-format instructions

opcode	address
6 bits	26 bits

32 bits

j 1028

1028/4 = 257
We will know why later..

Representation in decimal →

2	**257**

Representation in binary →

001000	**00000000000000000100000001**

Check your Understanding

State the instruction format for each of the following instructions, and show their representation in decimal. Given that funct of add instruction is 32, the opcode for addi is 8, and the opcode for jump instruction is 2.

- add $s3, $s1, $s2
- addi $s4, $s3, 1
- j 1024

$s1 → R17
$s2 → R18
$s3 → R19
$s4 → R20

Solution

This instruction add \$s3, \$s1, \$s2 is in J-format

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

32 bits

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

0	**17**	**18**	**19**	**0**	**32**

Solution

This instruction addi $s4, $s3, 1 is in I-fomat

opcode	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

32 bits

addi $s4, $s3, 1

Representation in decimal →

8	**19**	**20**	**1**

Solution

This instruction j 1024 is in J-format

opcode	address
6 bits	26 bits

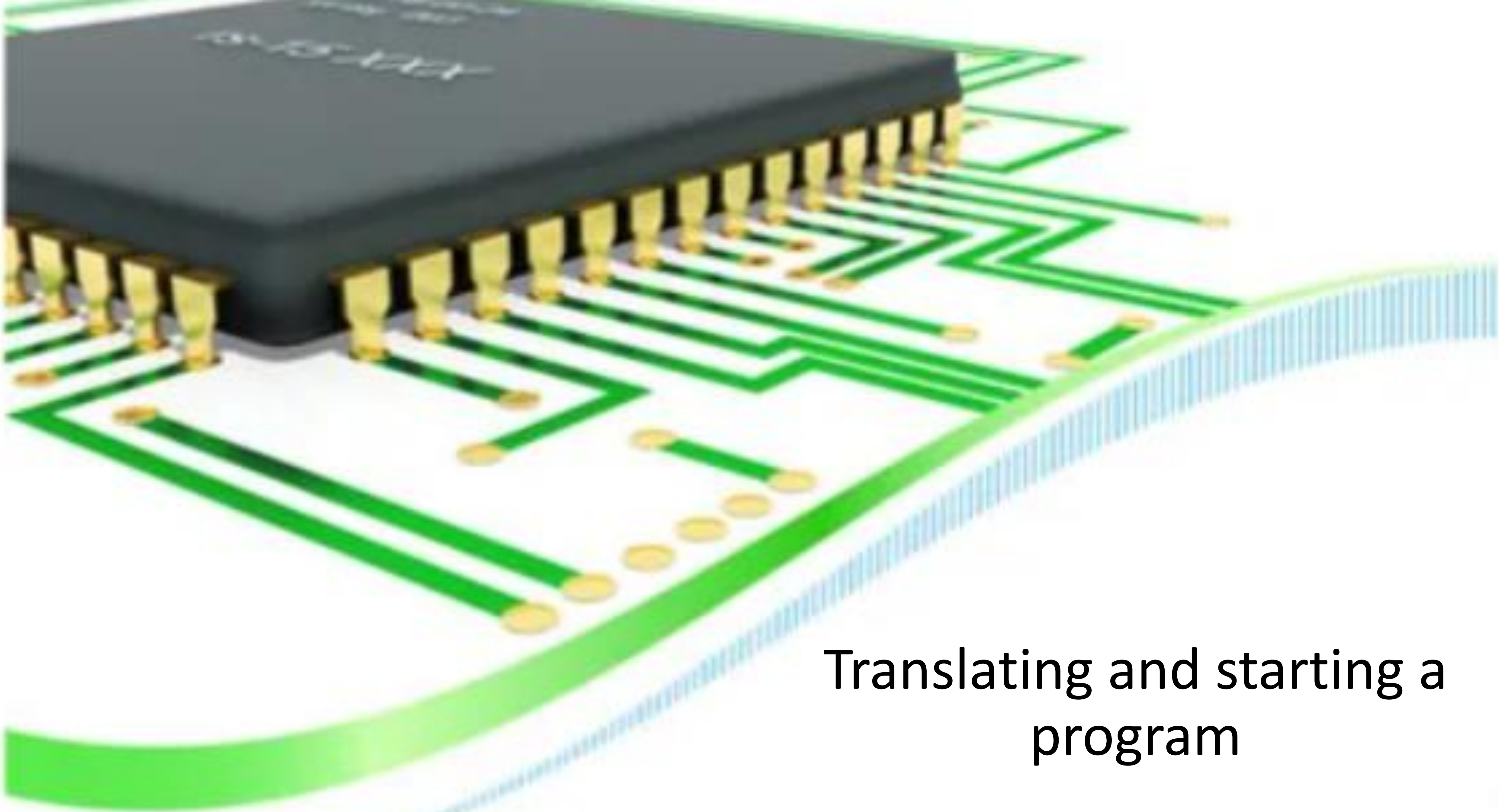
32 bits

j 1024

1024/4 = 256
We will know why later..

Representation in decimal →

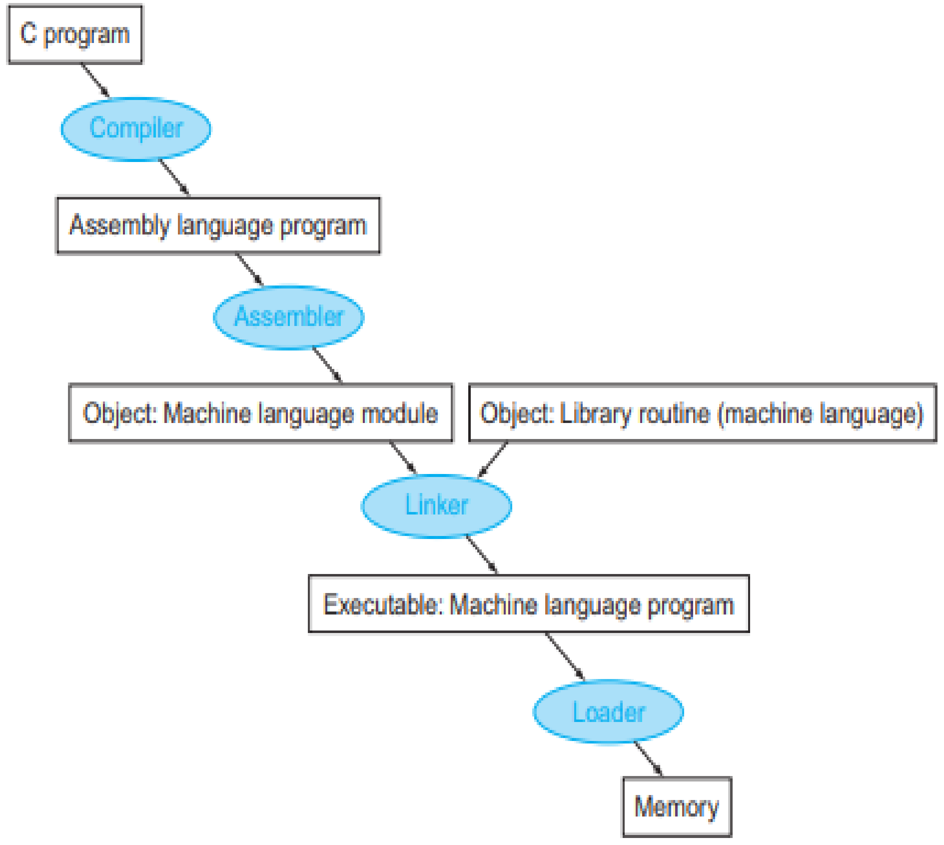
2	**256**



Translating and starting a program

Translating and starting a C program

- There are four steps in transforming a C program from a file on disk into a program running on a computer:
    1. Compiler
    2. Assembler
    3. Linker
    4. Loader



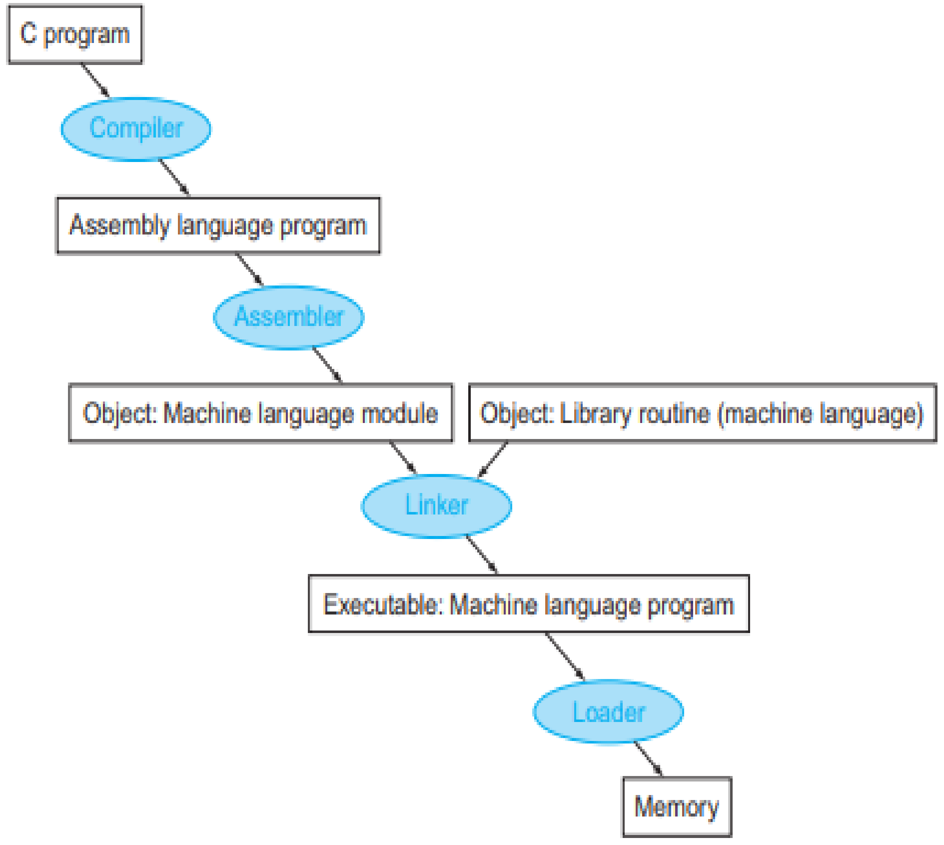
A translation hierarchy for C program

Translating and starting a C program

Compiler:

- The compiler transforms the C program into an assembly language program.

- An assembly language is a symbolic language that can be translated into binary machine language.

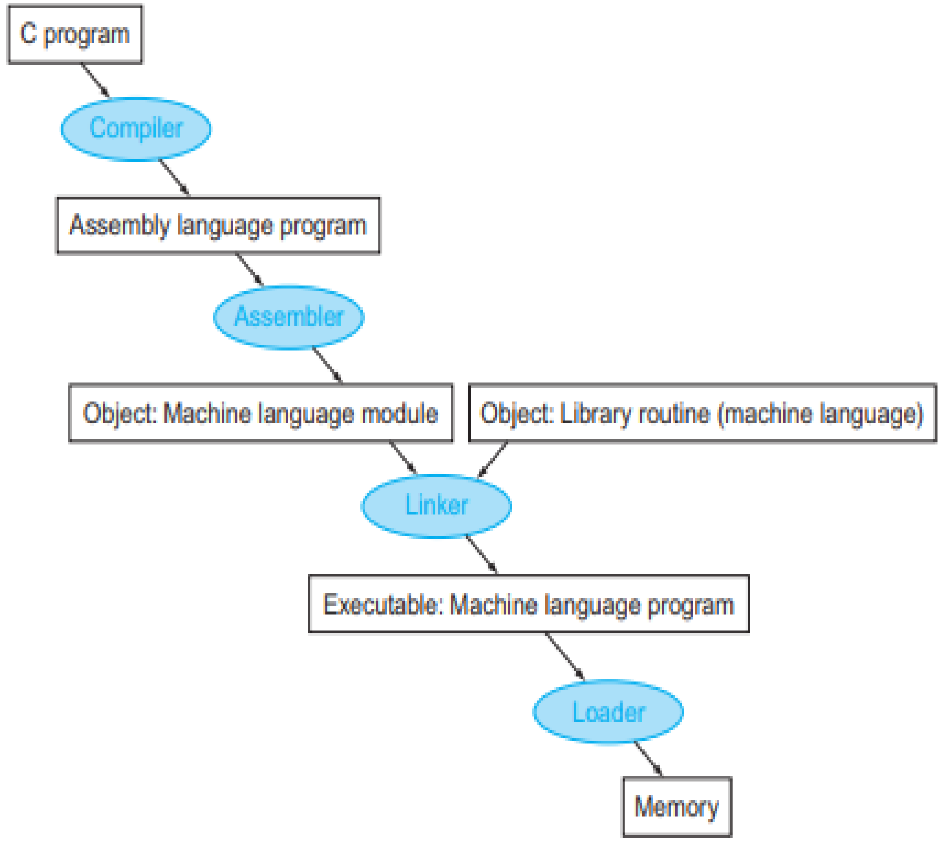


A translation hierarchy for C program

Translating and starting a C program

Assembler:

- The assembler turns the assembly language program into an object file, which is a combination of machine language instructions, data, and information needed to place instruction properly in memory.



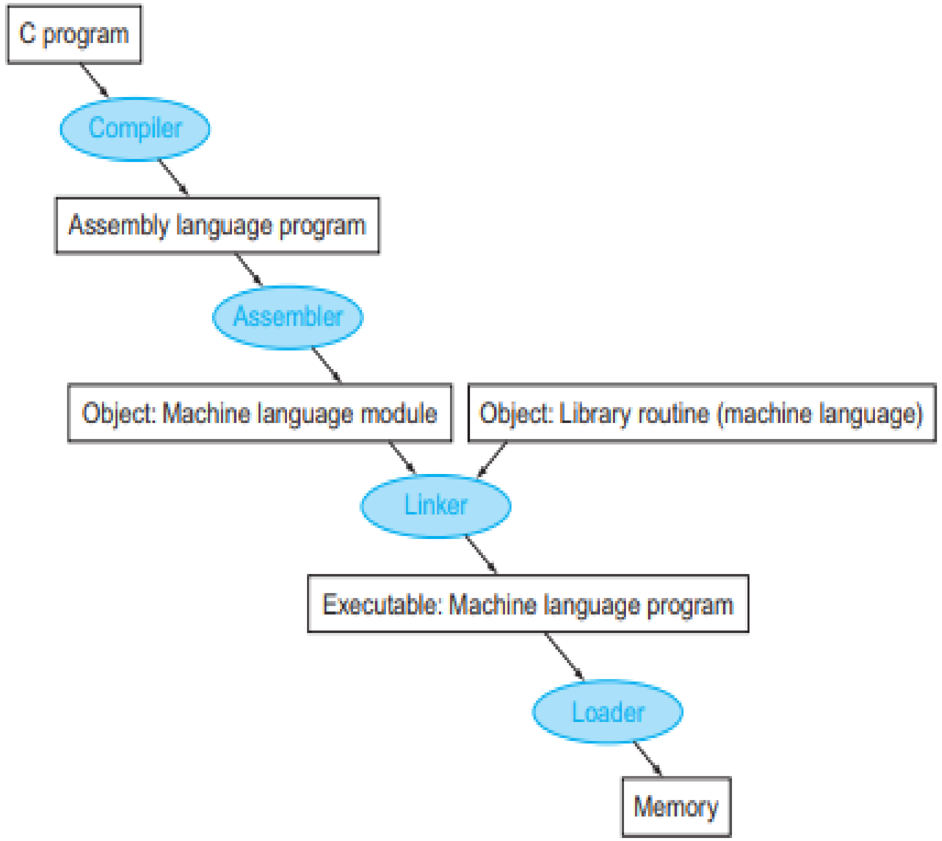
A translation hierarchy for C program

Translating and starting a C program

Assembler:

- To produce the binary version of each instruction in the assembly language program, the assembler must determine the addresses corresponding to all labels.

Assemblers keep track of labels in a *symbol table.*



A translation hierarchy for C program

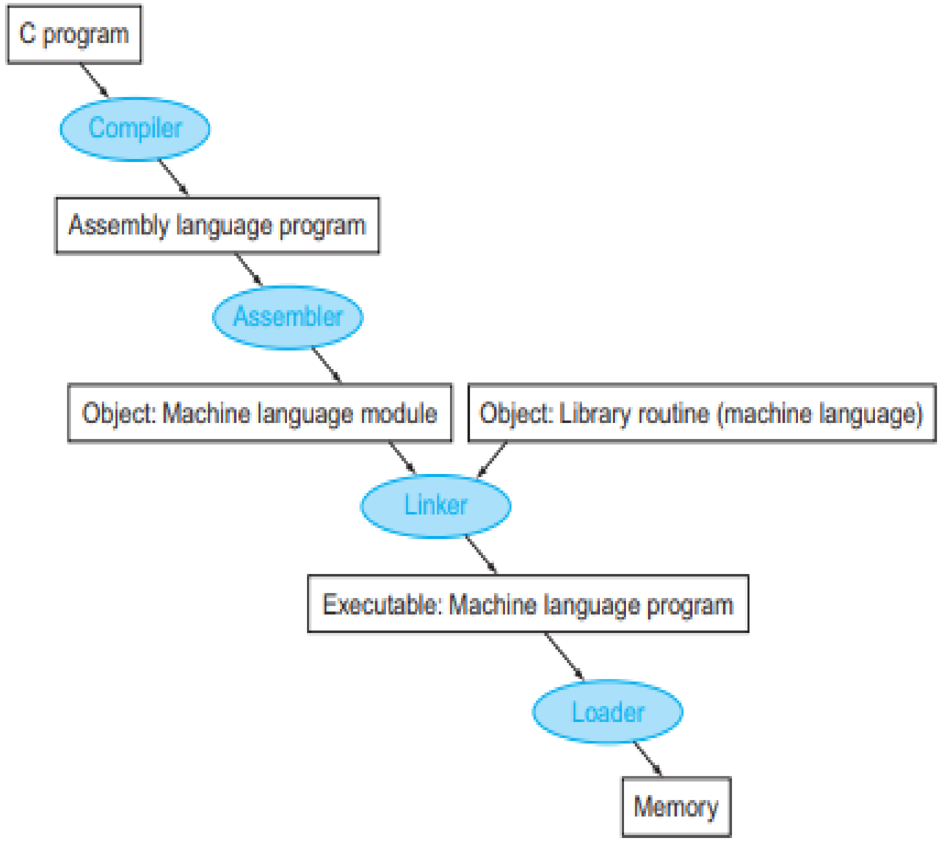
Translating and starting a C program

Linker:

- A systems program that combines independently assembled machine language programs and resolves all undefined labels into an executable file.

There are three steps for the linker:

1. Place code and data modules symbolically in memory.
2. Determine the addresses of data and instruction labels.
3. Patch both the internal and external references.

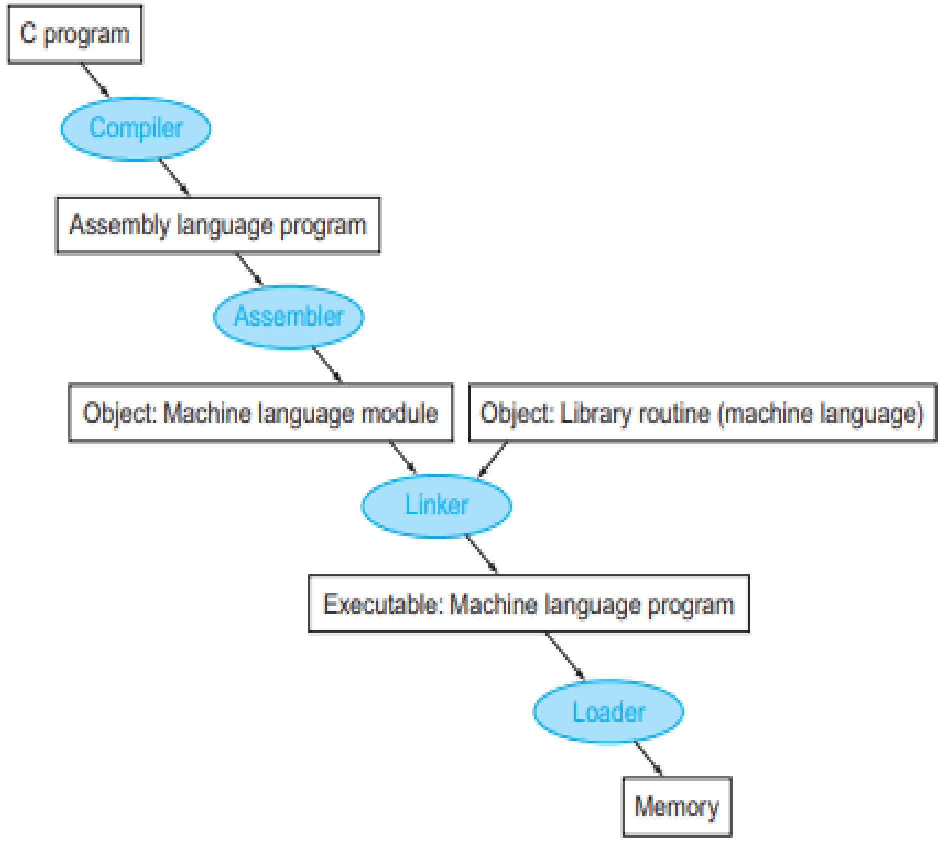


A translation hierarchy for C program

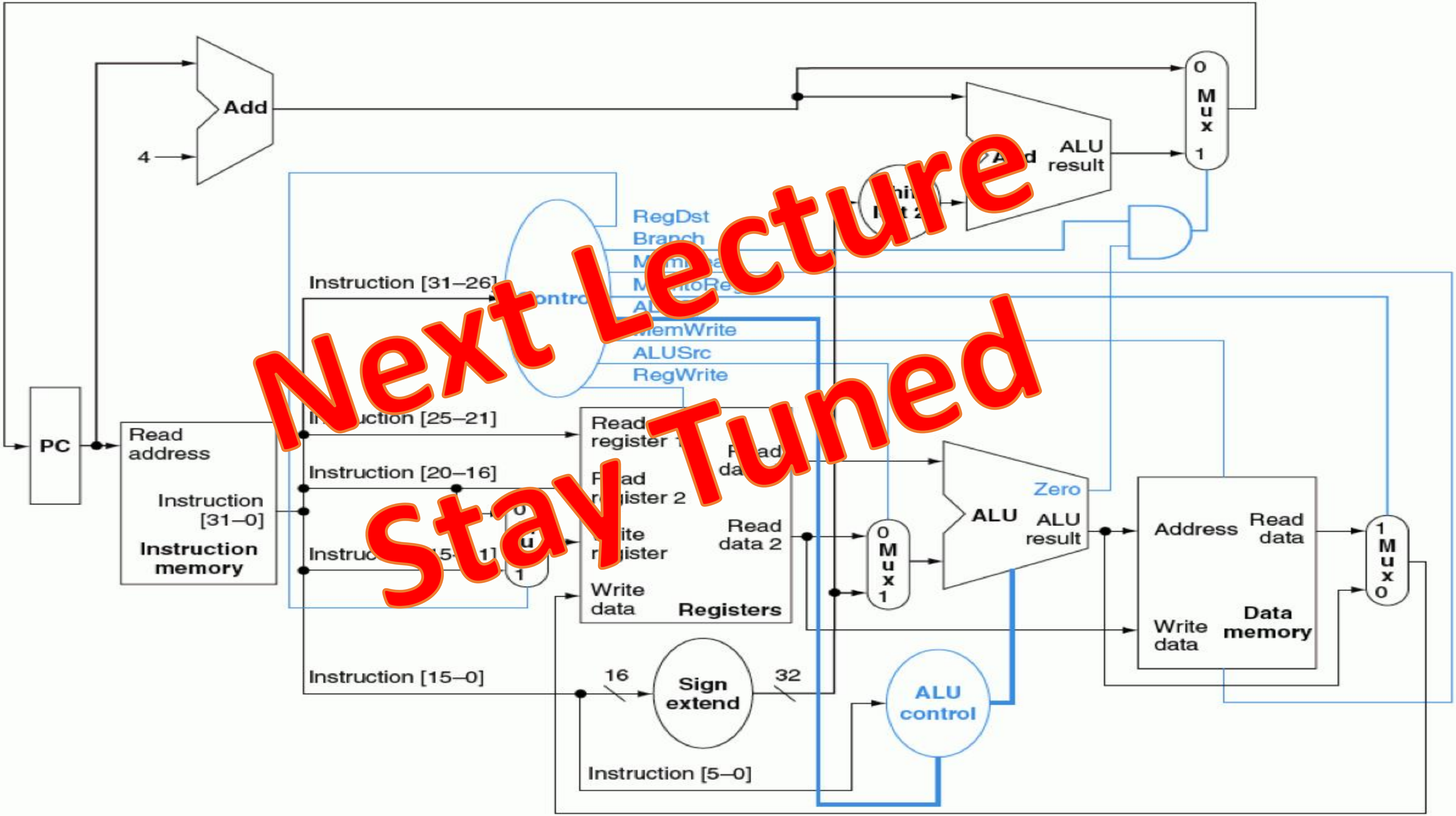
Translating and starting a C program

Loader:

- A systems program that places an object program in main memory so that it is ready to execute.



A translation hierarchy for C program



Next Lecture
Stay Tuned
PC
Add
4
ALU result
Mux
0
1
RegDst
Branch
MemWrite
ALUSrc
RegWrite
Instruction [31–26]
Read address
Instruction [31–0]
Instruction memory
Instruction [20–16]
Write data
Registers
Read data 2
ALU
Zero
ALU result
Address
Read data
Write data
Data memory
Instruction [15–0]
16
Sign extend
32
ALU control
Instruction [5–0]

Thank You