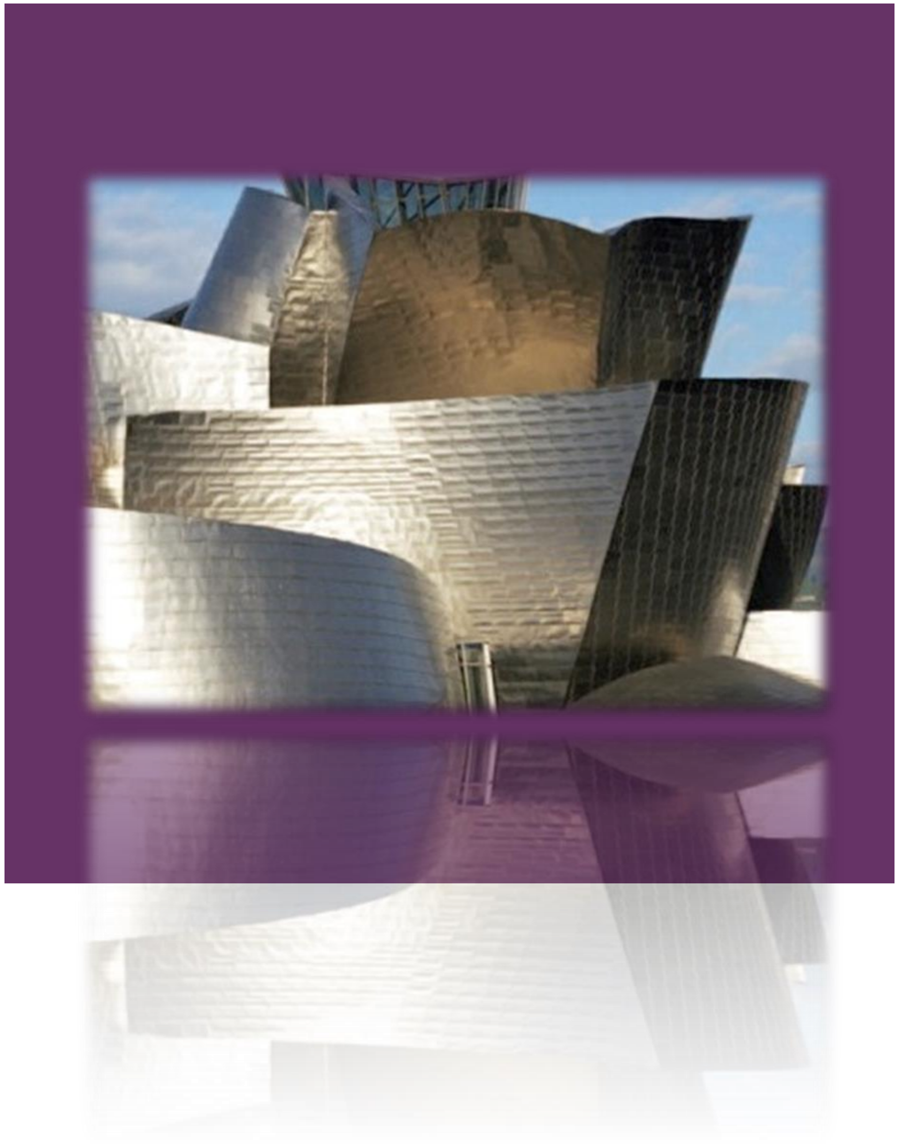


SECR2033 Computer Organization and Architecture

Lecture slides prepared by "Computer Organization and Architecture", 9/e, by William Stallings, 2013.

Module 4
Instruction Set Architecture (ISA)

Objectives:

- To provide a more detailed look at machine instruction sets.
- To look at different instruction types and operand types, and how instructions access data in memory.
- To understanding how instruction sets are designed and how their function can help to understand the more intricate details of the architecture of the machine itself.

Module 4
Instruction Set Architecture (ISA)

Module 4
Instruction Set Architecture (ISA)

- ❑ Overview
- ❑ Hierarchy of Computer Languages
- ❑ General Concepts:
  x86 Processor Architecture
- ❑ Design Decisions for Instruction Sets
- ❑ ISA Level

Overview

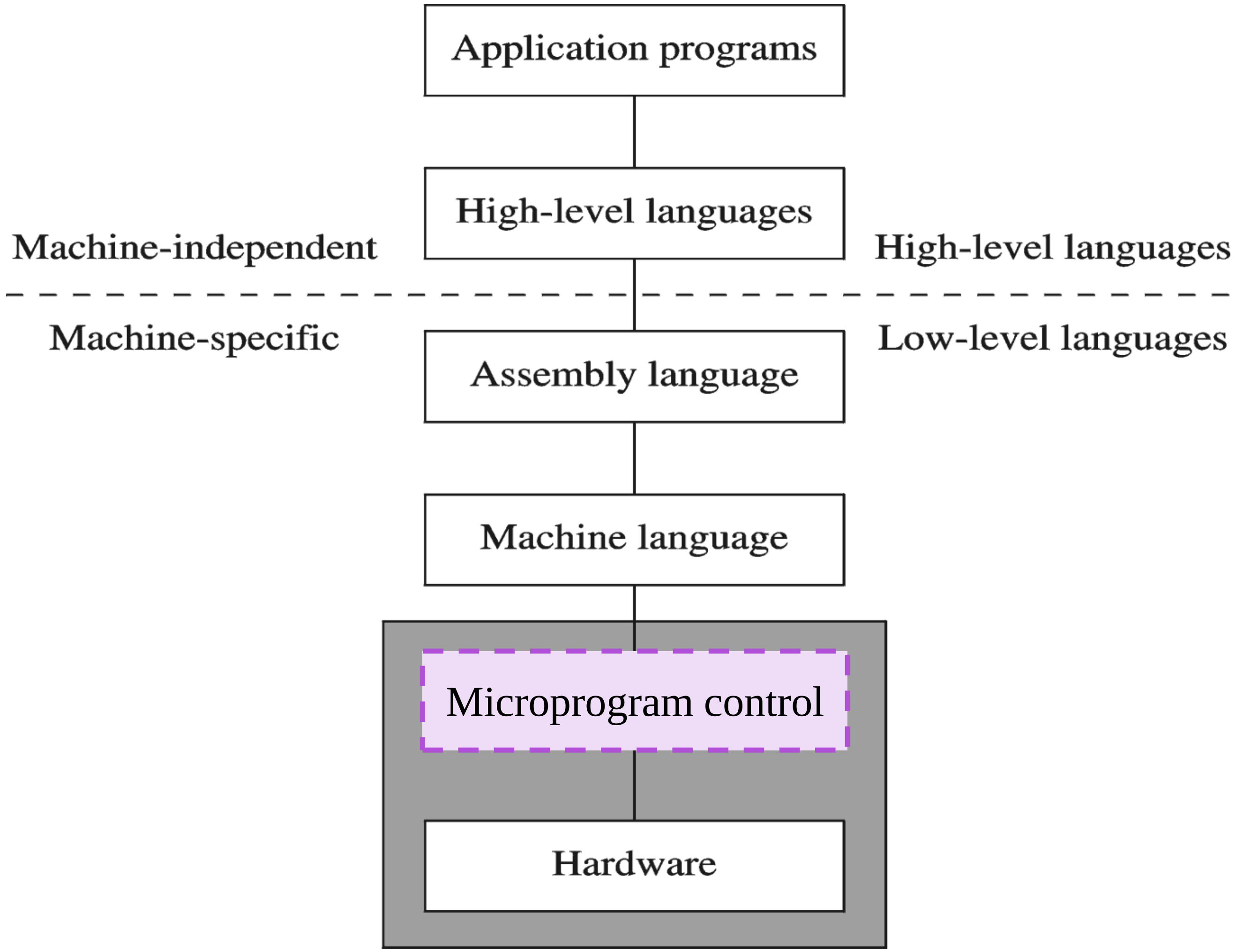
ALU (Arithmetic Logic Unit)

- One boundary where the computer designer and the computer programmer can view the same machine is the machine instruction set.
- Implementing the processor is a task that in large part involves implementing the machine instruction set.
- The user who chooses to program in machine language (actually, in assembly language) becomes aware of the register and memory structure, the types of data directly supported by the machine, and the functioning of the ALU.

Some important questions to ask:

- *What is assembly language?*
- *Why learn assembly language?*
- *What is machine language?*
- *How is assembly related to machine language?*
- *What is an assembler?*
- *How is assembly related to high-level language?*
- *Is assembly language portable?*

Hierarchy of Computer Languages



Assembly and Machine Languages

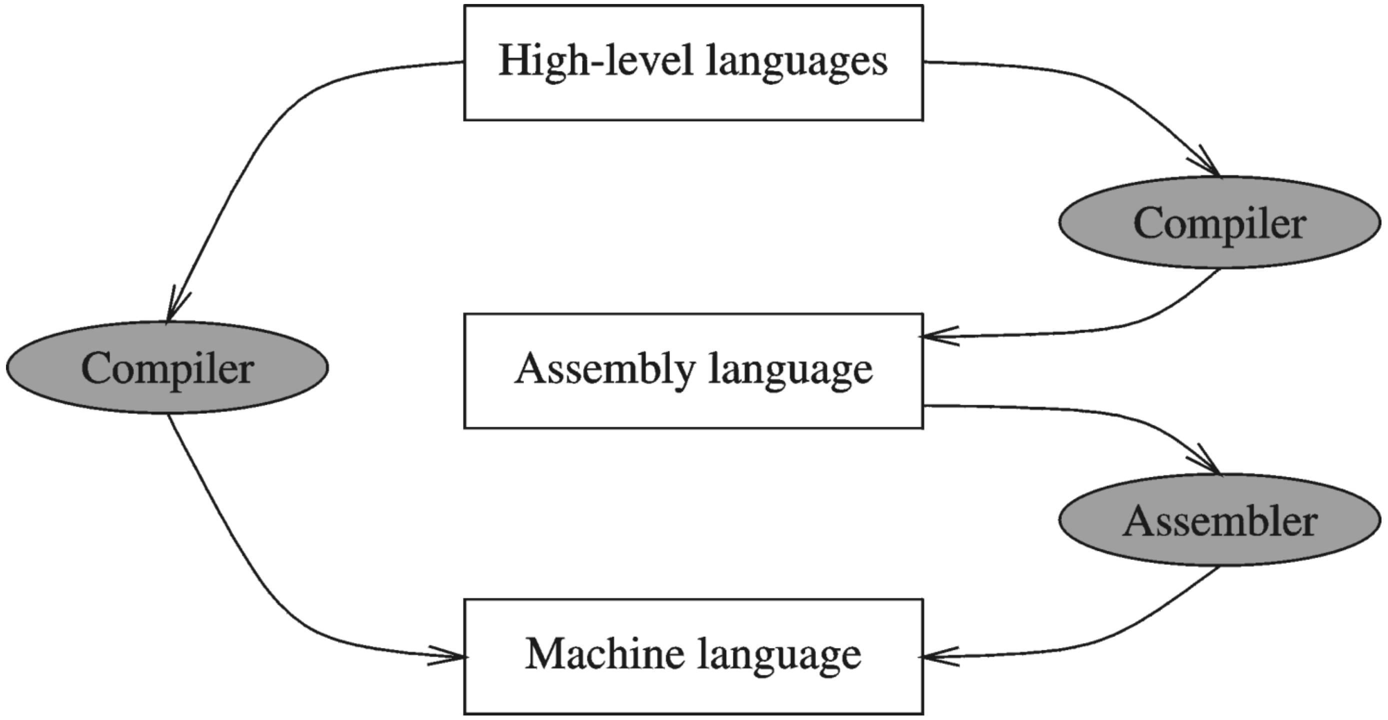
Machine language:

- Native to a processor: executed directly by hardware.
- Instructions consist of binary code: 1s and 0s

Assembly language:

- A programming language that uses *symbolic names* to represent operations, registers and memory locations.
- Slightly higher-level language.
- Readability of instructions is better than machine language.
- One-to-one correspondence with machine language instructions.

Compiler and Assembler



Compilers translate high-level programs to machine code:
→ either directly, or Indirectly via an *assembler*.

Assemblers translate assembly to machine code.

High-Level Languages: Advantages

- Program development is faster: fewer instructions to code.
- Program maintenance is easier: same reasons as above.

However, *Assembly language* programs are *not portable*.

- Programs are portable:
  - ❑ Contain few machine-dependent details → Can be used with little or no modifications on different machines.
  - ❑ Compiler translates to the target machine language.

Why Assembly Languages?

Accessibility to system hardware:

- ❑ Assembly language is useful for implementing system software.
- ❑ Also useful for small embedded system applications.

Space and Time efficiency:

- ❑ Understanding sources of program inefficiency.
- ❑ Tuning program performance.
- ❑ Writing compact code.

Writing assembly programs gives the computer designer the needed deep understanding of the instruction set and how to design one.

To be able to write compilers for HLLs, we need to be expert with the machine language. Assembly programming provides this experience.

General Concepts:
X86 Processor Architecture

- This section describes the architecture of the x86 processor family and its host computer system from a programmer's point of view.
- *Assembly language* is a great tool for learning how a computer works, and it requires you to have a working knowledge of computer hardware.
- The concepts in this section will help to understand the assembly language code that be written.

Basic Microcomputer Design

(Section 5.2)

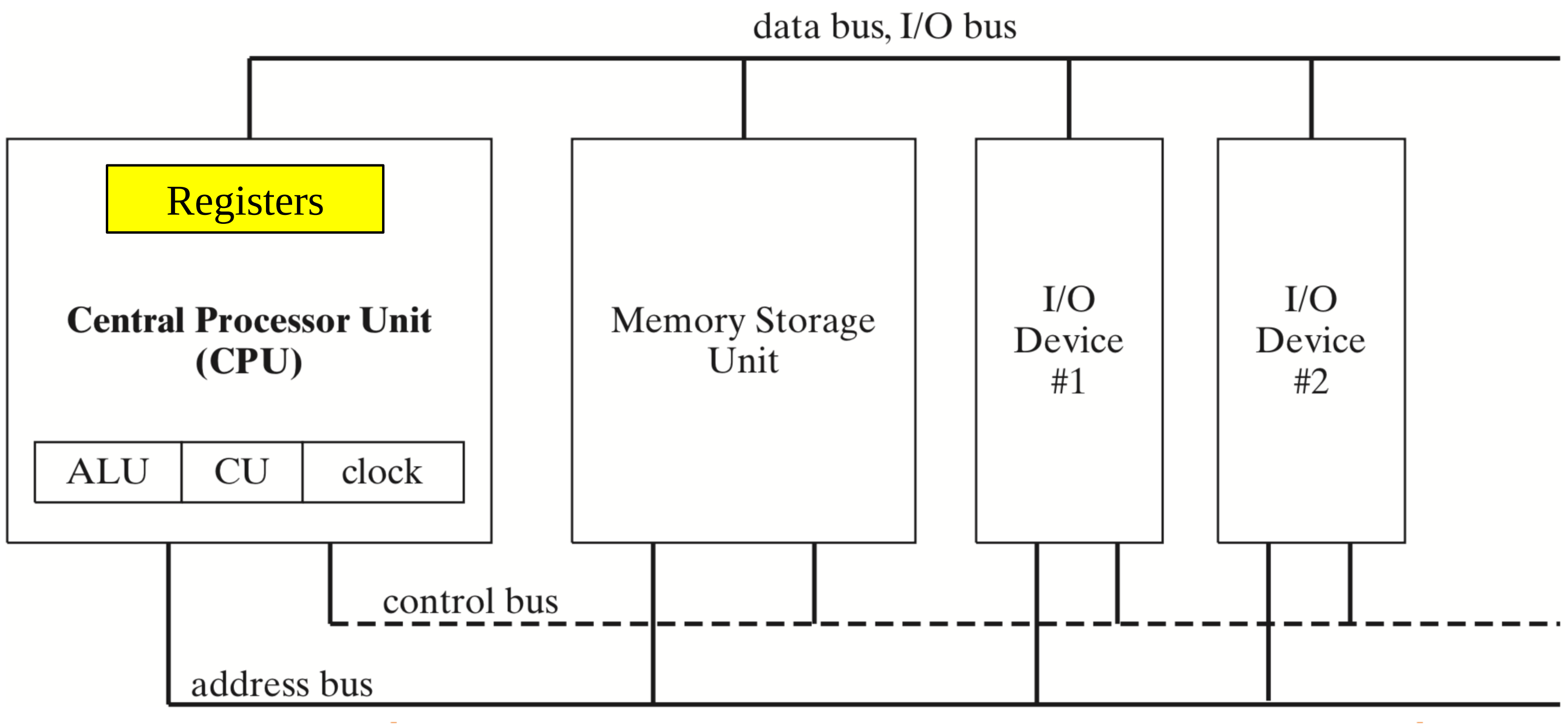


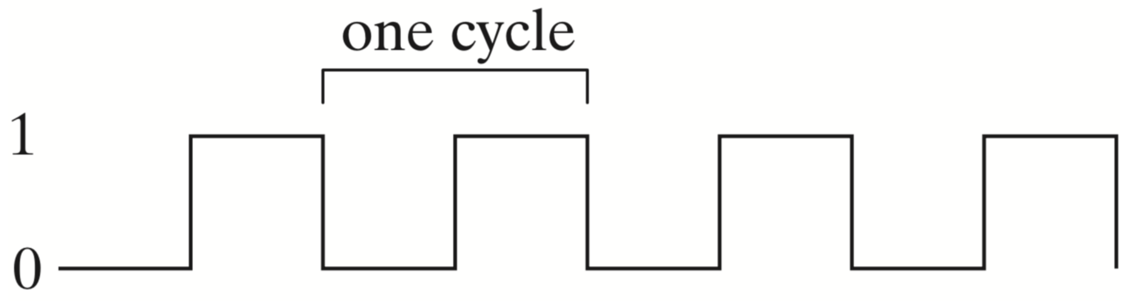
Figure: Block diagram of a microcomputer.

- The CPU does the *calculations* and *logic operations* → contains a limited number of storage locations as follow:

Elements	Description
Clock	○ synchronizes the internal CPU operations.
Control Unit (*CU*)	○ coordinates sequence of execution steps.
Arithmetic Logic Unit (*ALU*).	○ performs arithmetic and bitwise processing.

Clock

- synchronizes all CPU and BUS operations.
- machine (clock) cycle measures time of a single operation.
- clock is used to trigger events.

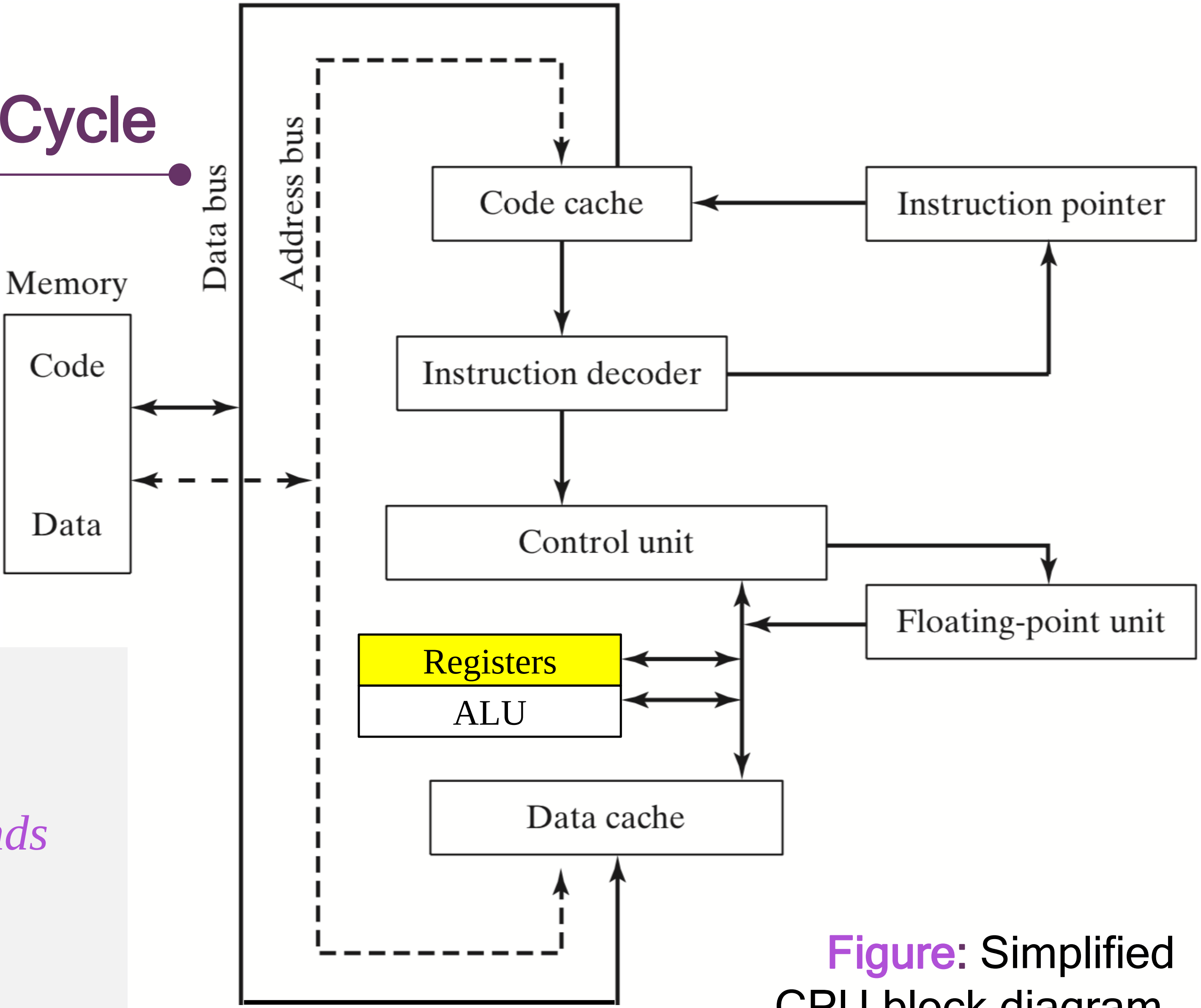


Video 2 - Computer Clock

Remember this video from Module 1?

- What is clock in a computer? Please watch this YouTube video
- https://www.youtube.com/watch?v=Z5JC9Ve1sfI

Instruction Execution Cycle



- *Fetch*
- *Decode*
- *Fetch operands*
- *Execute*
- *Store output*

Figure: Simplified CPU block diagram.

a clock cycle begins as the clock signal changes from high to low

Reading From Memory

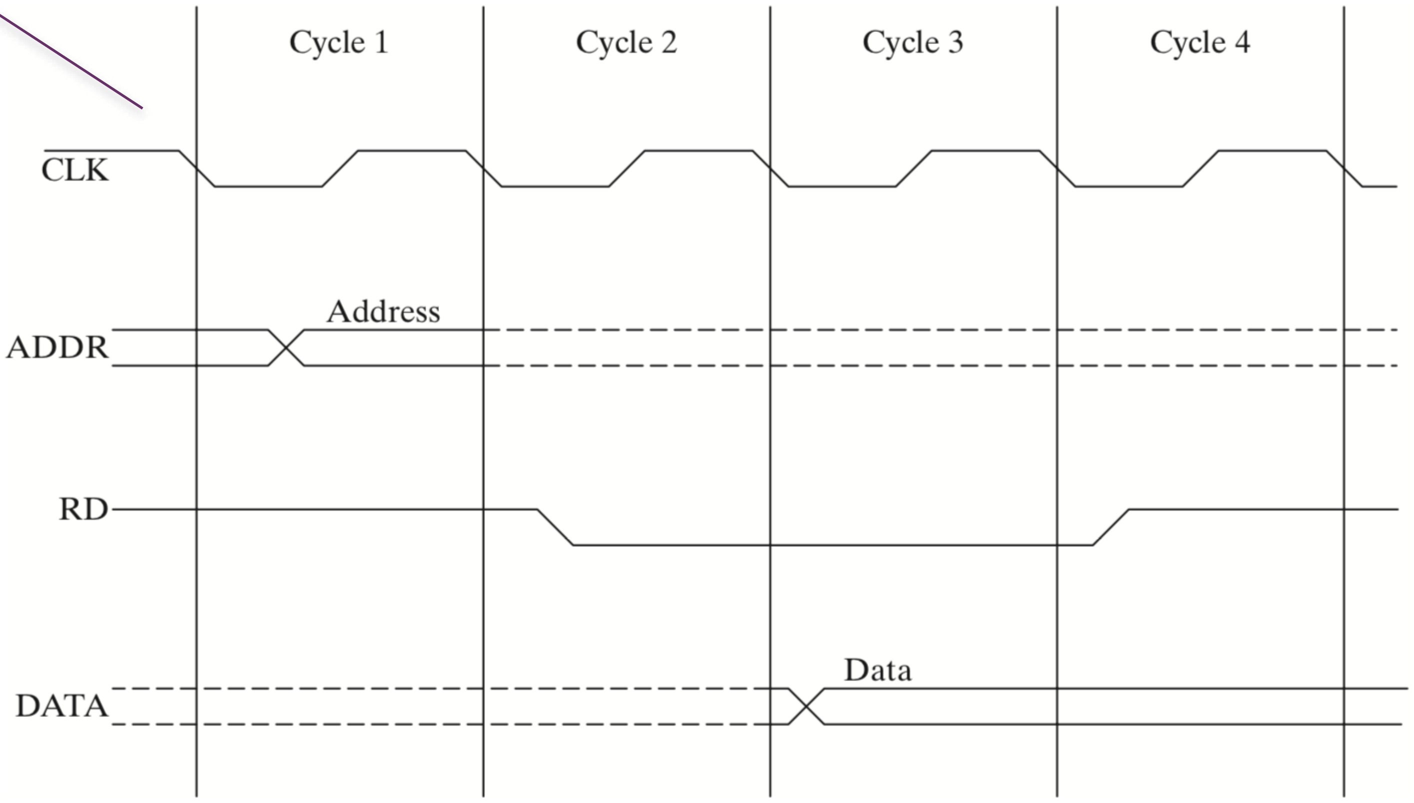
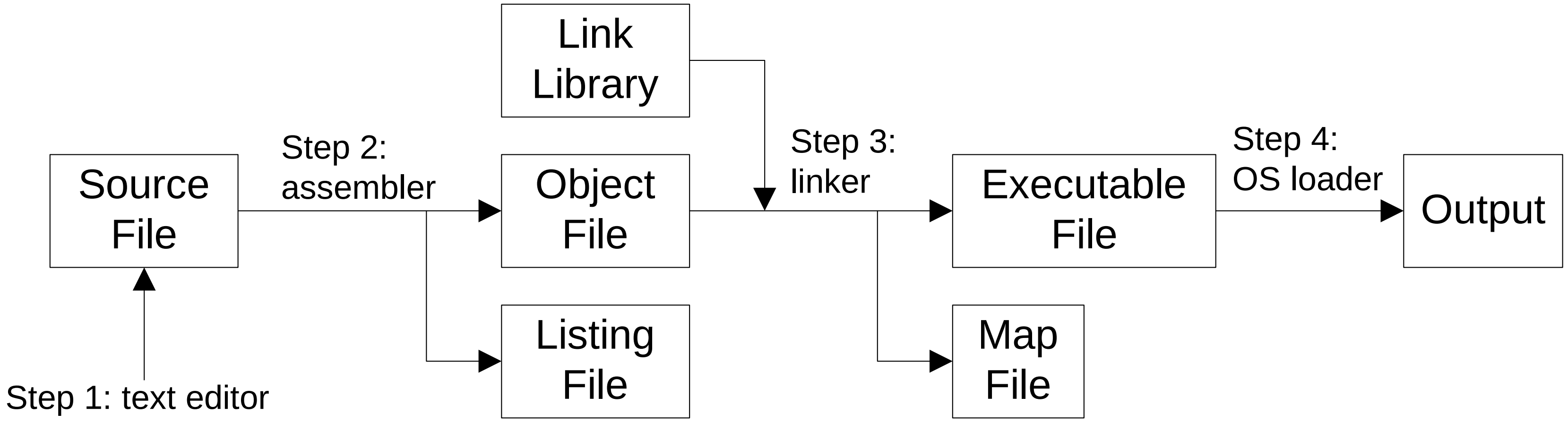


Figure: Memory read cycle.

- Multiple machine cycles are required when reading from memory, because it responds much more slower than the CPU.
- The steps are:
  - **Cycle 1**: address placed on address bus (ADDR).
  - **Cycle 2**: *Read Line* (RD) set low (0) to notify memory that a value is to be read.
  - **Cycle 3**: CPU waits one cycle for memory to respond.
  - **Cycle 4**: *Read Line* (RD) goes to high (1), indicating that the data is on the data bus (DATA).

How Program Run?

Assembling, Linking, and Running Programs

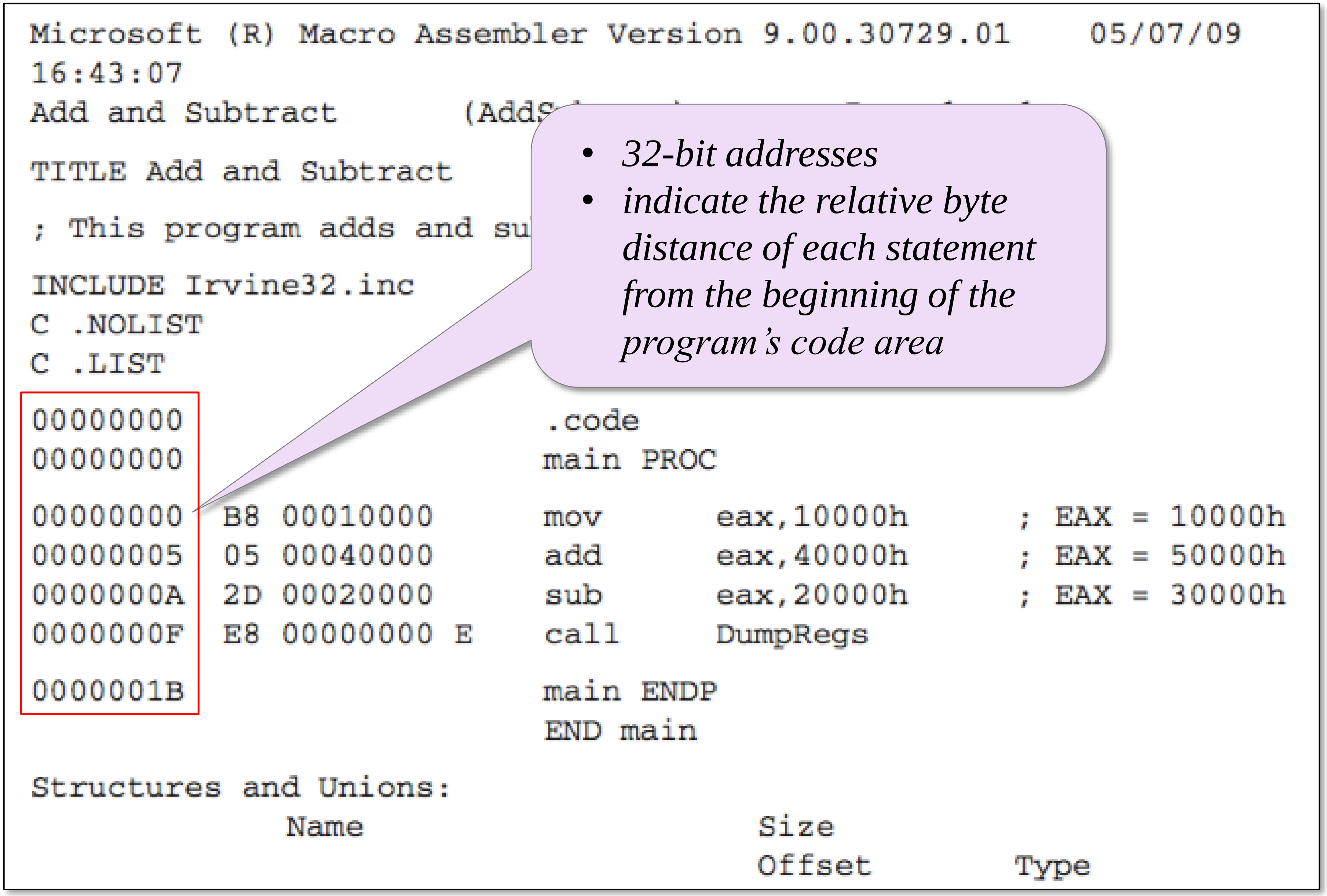


- The diagram describes the steps from creating a source program through executing the compiled program.
- If the source code is modified, Steps 2 through 4 must be repeated.

Listing File

- Use it to see how your program is compiled.
- Contains:
  - *source code*
  - *addresses*
  - *object code (machine language)*
  - *segment names*
  - *symbols (variables, procedures, and constants)*
- **Example**: `addSub.lst`

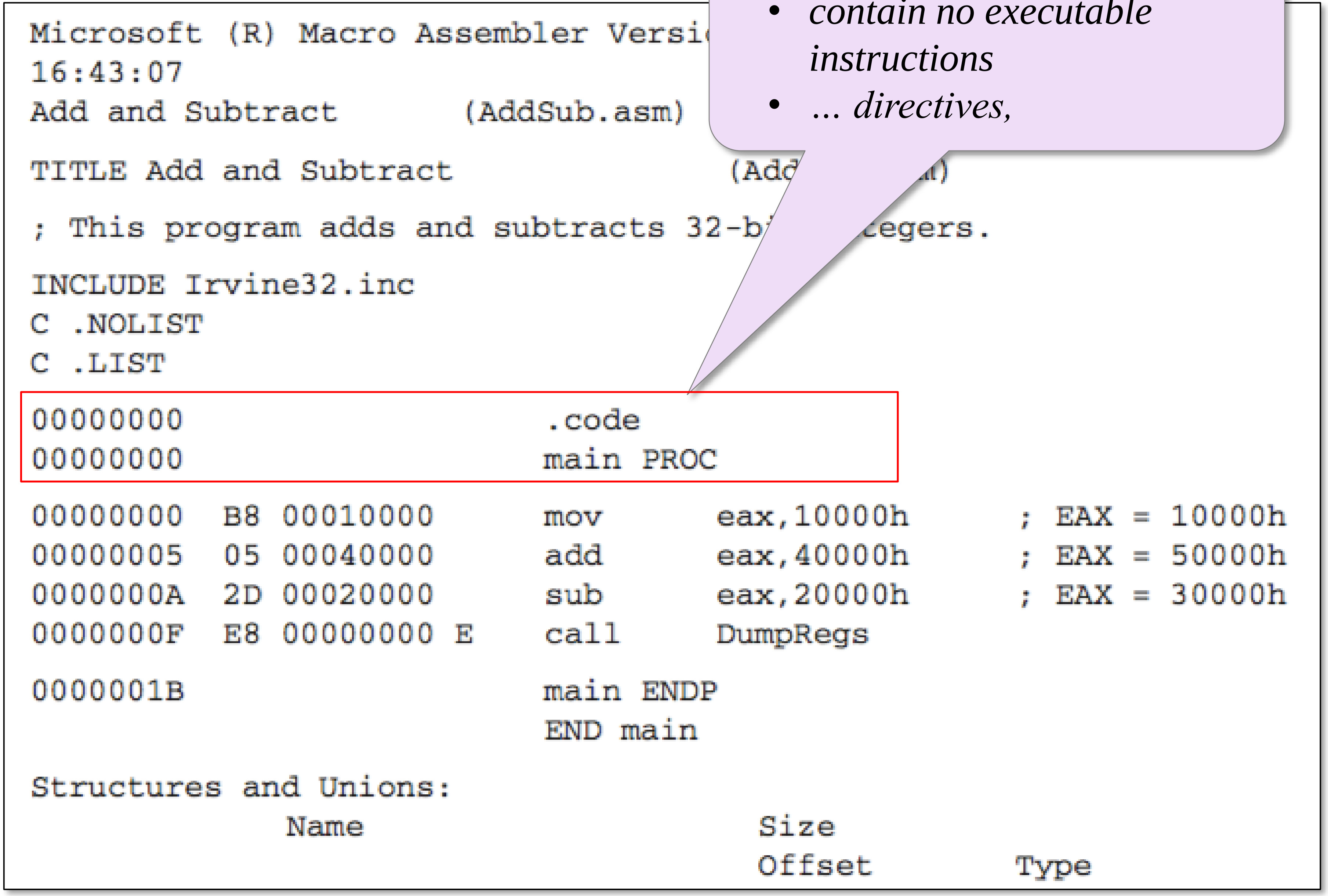
```
Microsoft (R) Macro Assembler Version 9.00.30729.01    05/07/09
16:43:07
Add and Subtract       (AddSub.asm)          Page 1 - 1

TITLE Add and Subtract                 (AddSub.asm)

; This program adds and subtracts 32-bit integers.

INCLUDE Irvine32.inc
C .NOLIST
C .LIST

00000000                          .code
00000000                          main PROC

00000000   B8 00010000            mov       eax,10000h        ; EAX = 10000h
00000005   05 00040000            add       eax,40000h        ; EAX = 50000h
0000000A   2D 00020000            sub       eax,20000h        ; EAX = 30000h
0000000F   E8 00000000 E          call      DumpRegs

0000001B                          main ENDP
                                  END main

Structures and Unions:
              Name                          Size
                                            Offset         Type
```



```
Microsoft (R) Macro Assembler Version 9.00.30729.01    05/07/09
16:43:07
Add and Subtract          (AddS

TITLE Add and Subtract

; This program adds and su

INCLUDE Irvine32.inc
C .NOLIST
C .LIST
00000000                      .code
00000000                      main PROC

00000000  B8 00010000         mov     eax,10000h      ; EAX = 10000h
00000005  05 00040000         add     eax,40000h      ; EAX = 50000h
0000000A  2D 00020000         sub     eax,20000h      ; EAX = 30000h
0000000F  E8 00000000 E       call    DumpRegs

0000001B                      main ENDP
                              END main

Structures and Unions:
             Name                         Size
                                          Offset        Type
```

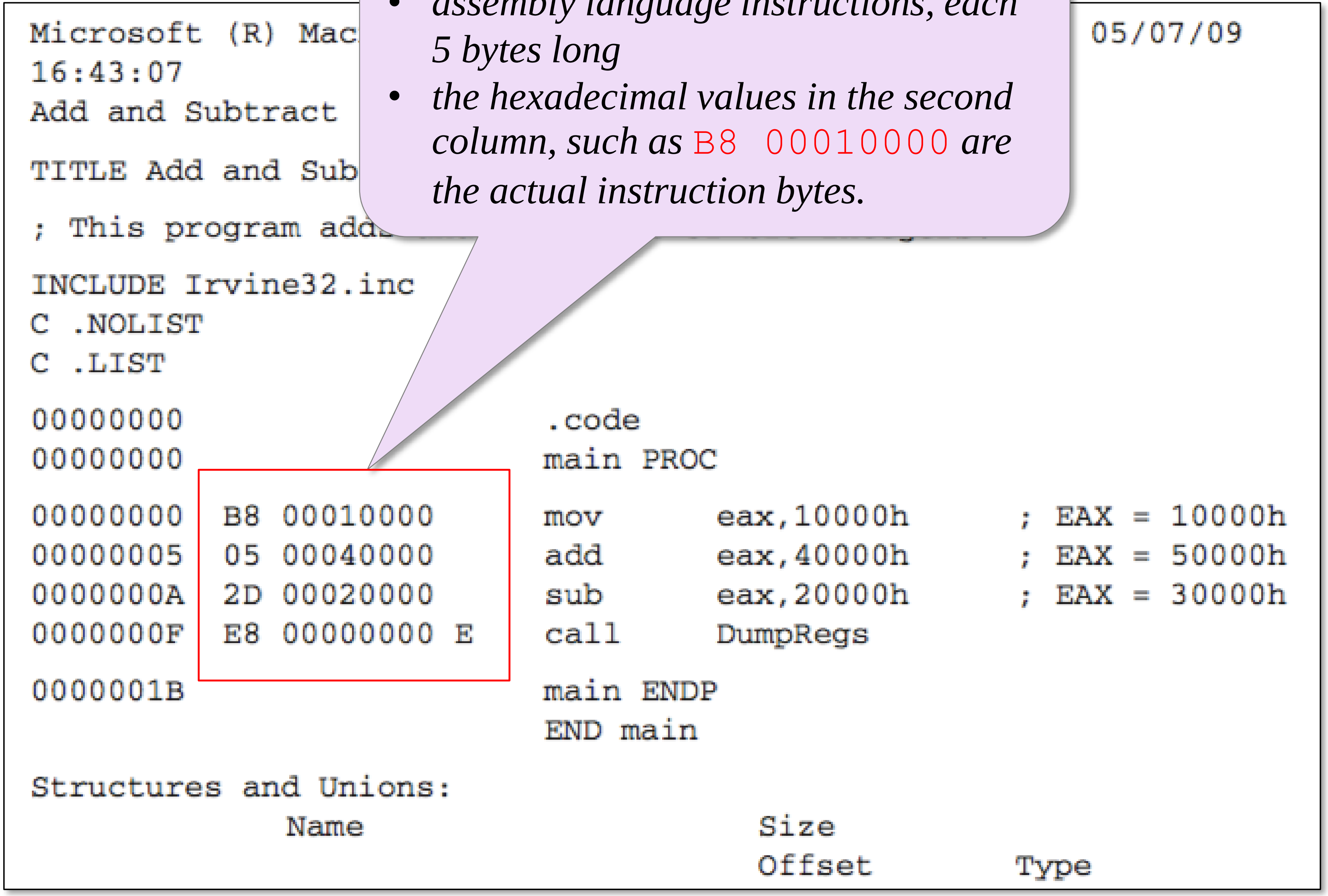
- *32-bit addresses*
- *indicate the relative byte distance of each statement from the beginning of the program's code area*

- *contain no executable instructions*
- *… directives,*



```
Microsoft (R) Macro Assembler Versi
16:43:07
Add and Subtract        (AddSub.asm)

TITLE Add and Subtract                    (Add        m)

; This program adds and subtracts 32-bi     tegers.

INCLUDE Irvine32.inc
C .NOLIST
C .LIST

00000000                          .code
00000000                          main PROC

00000000  B8 00010000             mov      eax,10000h      ; EAX = 10000h
00000005  05 00040000             add      eax,40000h      ; EAX = 50000h
0000000A  2D 00020000             sub      eax,20000h      ; EAX = 30000h
0000000F  E8 00000000 E           call     DumpRegs

0000001B                          main ENDP
                                  END main

Structures and Unions:
               Name                         Size
                                            Offset         Type
```

- *assembly language instructions, each 5 bytes long*
- *the hexadecimal values in the second column, such as* `B8 00010000` *are the actual instruction bytes.*

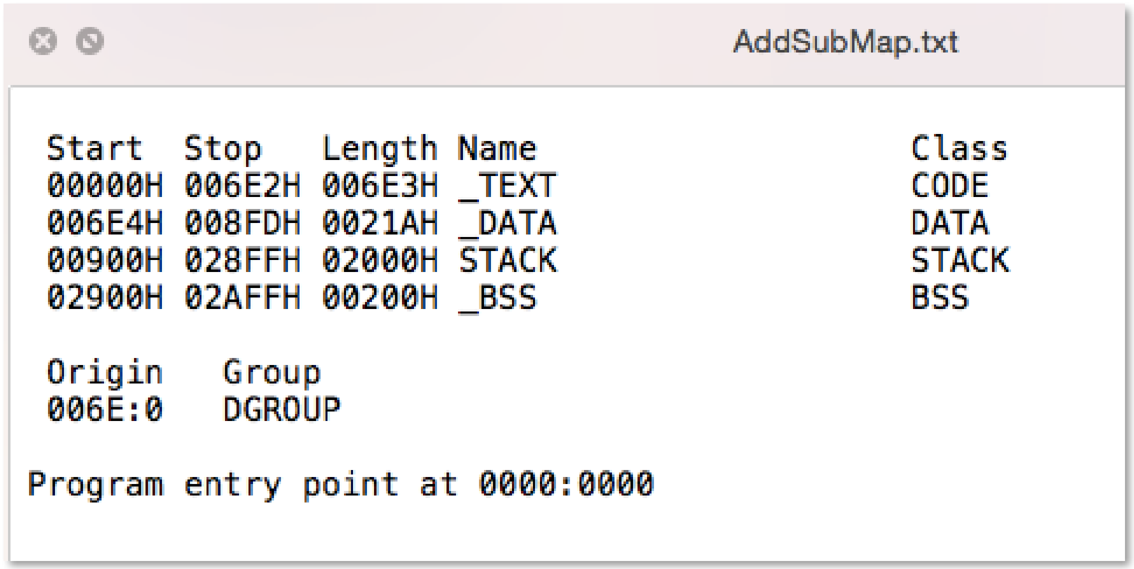


```
Microsoft (R) Mac                                          05/07/09
16:43:07
Add and Subtract

TITLE Add and Sub

; This program add

INCLUDE Irvine32.inc
C .NOLIST
C .LIST

00000000                    .code
00000000                    main PROC

00000000  B8 00010000       mov      eax,10000h      ; EAX = 10000h
00000005  05 00040000       add      eax,40000h      ; EAX = 50000h
0000000A  2D 00020000       sub      eax,20000h      ; EAX = 30000h
0000000F  E8 00000000 E     call     DumpRegs

0000001B                    main ENDP
                            END main

Structures and Unions:
             Name                         Size
                                          Offset        Type
```

Map File

- Information about each program segment:
  - *starting address*
  - *ending address*
  - *size*
  - *segment type*

- **Example**: `addSub.map`


AddSubMap.txt

```
Start  Stop   Length Name                      Class
00000H 006E2H 006E3H _TEXT                     CODE
006E4H 008FDH 0021AH _DATA                     DATA
00900H 028FFH 02000H STACK                     STACK
02900H 02AFFH 00200H _BSS                      BSS

Origin   Group
006E:0   DGROUP

Program entry point at 0000:0000
```

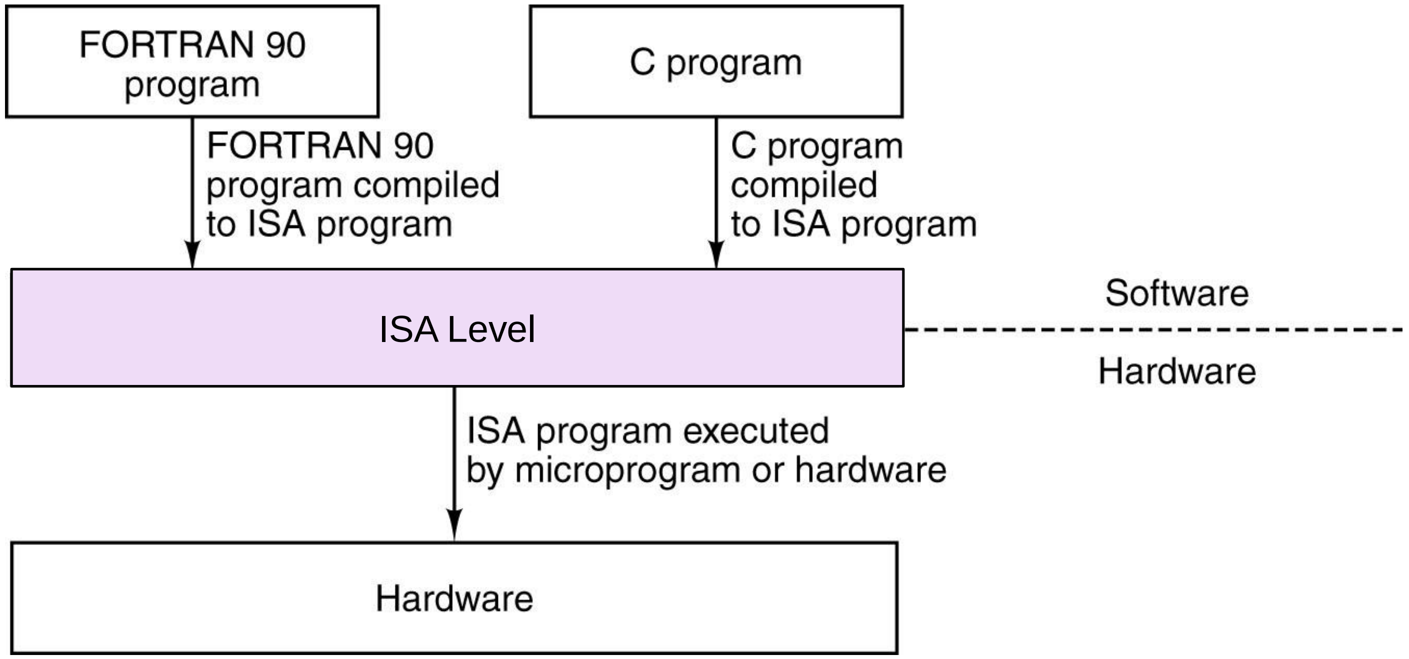
Design Decisions for Instruction Sets

- When a computer architecture is in the design phase, the instruction set format must be determined before many other decisions can be made.
- Selecting this format is often quite difficult because the instruction set must match the architecture.
- If the architecture is well designed, it could last for decades.

ISA Level

ISA (Instruction Set Architecture)

- ISA Level defines the interface between the compilers (high level language) and the hardware. It is the language that both of them understand.


FORTRAN 90 program
C program
FORTRAN 90 program compiled to ISA program
C program compiled to ISA program
ISA Level
Software
Hardware
ISA program executed by microprogram or hardware
Hardware

- Instruction Set Architectures (ISAs) are measured by several different factors, including:

(1) the <u>amount of space</u> a program requires;

(2) the <u>complexity</u> of the instruction set;

(3) the <u>length</u> of the instructions; and

(4) the <u>total number</u> of instructions.

Module 4
Instruction Set Architecture (ISA)

- ❑ Elements of a Machine Instruction
- ❑ Instruction Representation
- ❑ Instruction Types
- ❑ Number of Addresses
- ❑ Instruction Set Design

Elements of a Machine Instruction

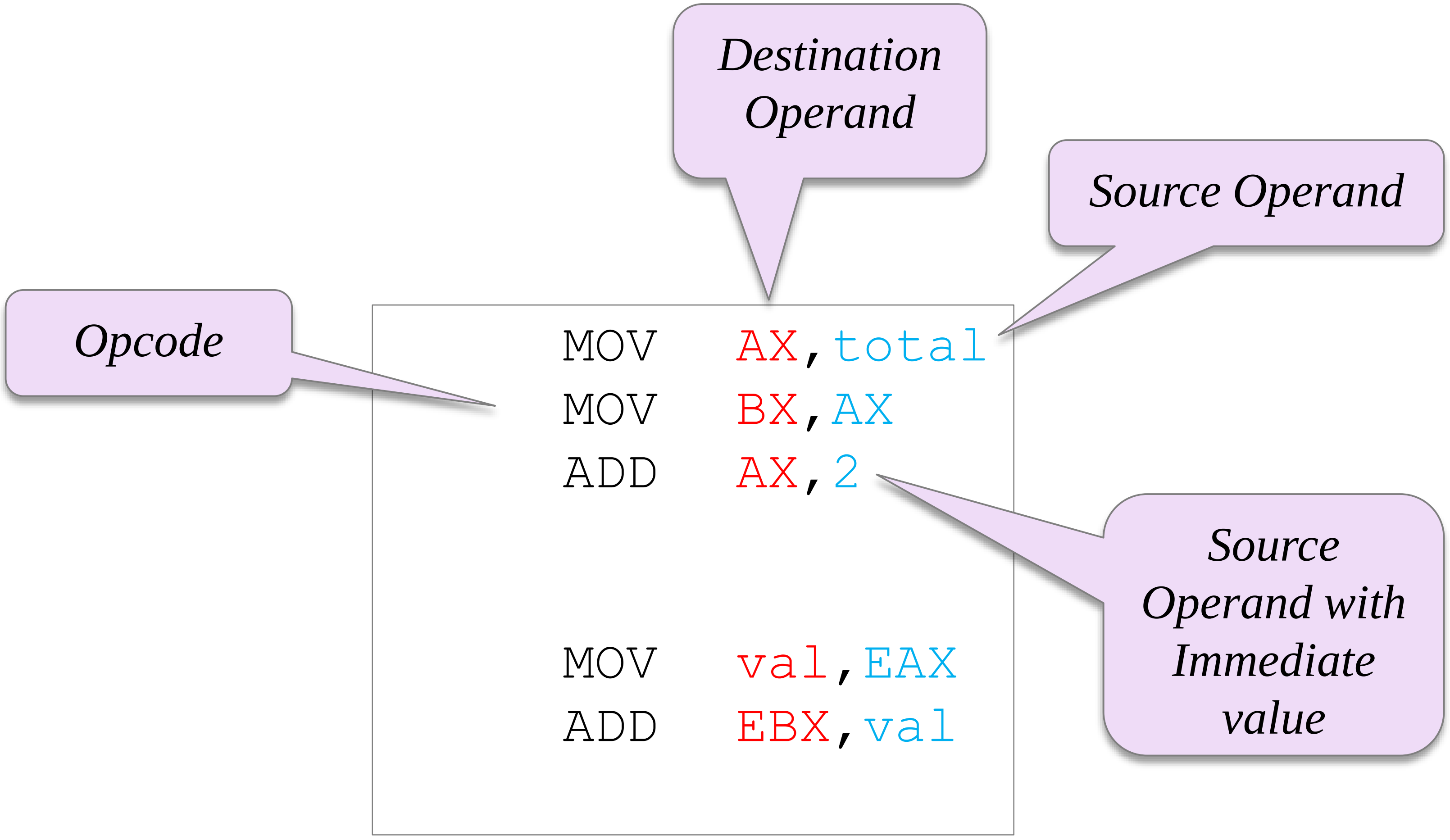
- The operation of the processor is determined by the instructions it executes, referred to as machine instructions or computer instructions.
- The collection of different instructions that the processor can execute is referred to as the processor's instruction set.
- Each instruction must contain the information required by the processor for execution.

Table: The elements of a machine instruction.

Elements	Description
Operation code *(Opcode)*	○ Specifies the operation to be performed a binary code (e.g., `MOV`, `ADD`, `SUB`).
Source operand reference	○ The operation may involve one or more source operands, that is, operands that are inputs for the operation.
Result operand reference *(Destination).*	○ The operation may produce a result.
Next instruction reference	○ This tells the processor where to fetch the next instruction after the execution of this instruction is completed.

Invisible

Example 1: Portion of an assembly language.



```
MOV    AX,total
MOV    BX,AX
ADD    AX,2


MOV    val,EAX
ADD    EBX,val
```

- *Source* and result operands (*Destination*) can be in one of four areas:

Section 4.4: Addressing Mode

  - Main or virtual memory :
    Memory address for both must be supplied.
  - Processor (CPU) registers : One or more registers that can be referenced by instructions.
  - Immediate : The value of the operand is contained in the field in the instruction executed.
  - I/O device – Instruction specifies the I/O module and device for the operation

Example 2:

Operand: Memory

Current Ins. : 0000
Next Ins. : 0001

```
0000  MOV   AX,TOTAL
0001  MOV   BX,AX
0002  ADD   AX,2
0003  TARGET
0004  CALL  READINT
0005  MOV   VAL,EAX
0006  ADD   EBX,VAL
0007  JMP   TARGET
```

Operand: Register

Operand: Immediate value

Operand: From I/O

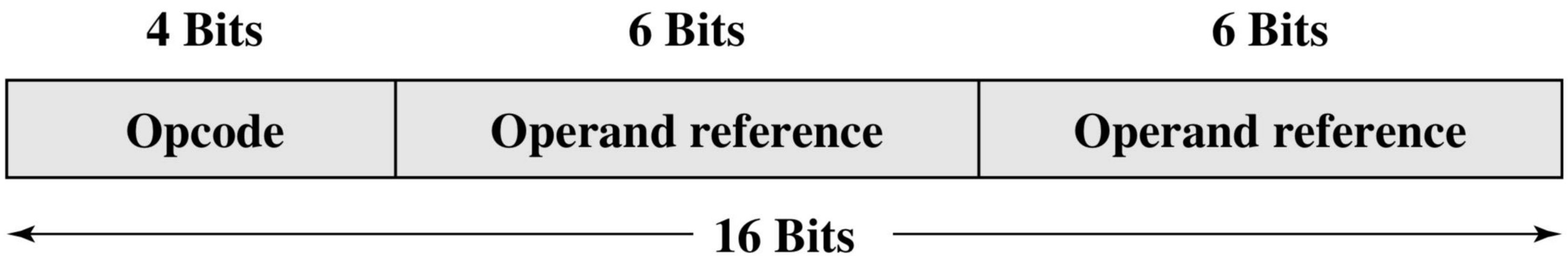
Current Ins. : 0007
Next Ins. : 0003

Next instruction is where TARGET is located = 0003

Instruction Representation

- Each instruction is represented by a sequence of bits that divided into fields, corresponding to the constituent elements of the instruction.

- **Example** of simple instruction format:



It is difficult for the programmer to deal with binary representations of machine instructions.

- Thus, it has become common practice to use a symbolic representation of machine instructions.
- Opcodes are represented by abbreviations, called *mnemonics,* that indicate the operation.
- Common **examples**:

ADD	Add
SUB	Subtract
MUL	Multiply
DIV	Divide
LOAD	Load data from memory
STOR	Store data to memory

- <u>Example</u>:

Instruction type	Opcode	Symbolic representation	Description
Data transfer	00001010	LOAD MQ,M(X)	Transfer contents of memory location X to register MQ

What the processor (CPU) see.

What the programmer see.

- During instruction execution:
  - an instruction is <u>read</u> into an Instruction Register (IR) in the processor.
  - The processor must be able to <u>extract</u> the data from the various instruction fields to <u>perform</u> the required operation.

Instruction Types

- Consider a high-level language instruction that could be expressed in a language such as C.
- **Example**:

```
Total = Total + stuff
```

A single C instruction may require 3 machine instructions; This is typical of the relationship between a high-level language and a machine language.

- In assembly language:

1) Load a register with the contents of memory (for `Total`).
2) Add the contents of memory (for `stuff`) to the register.
3) Store the content of the register to memory location (for `Total`).

- Translating the language:

English: `Total` is assigned the sum of `Total` and `stuff`.

High-Level Language: `Total = Total + stuff`

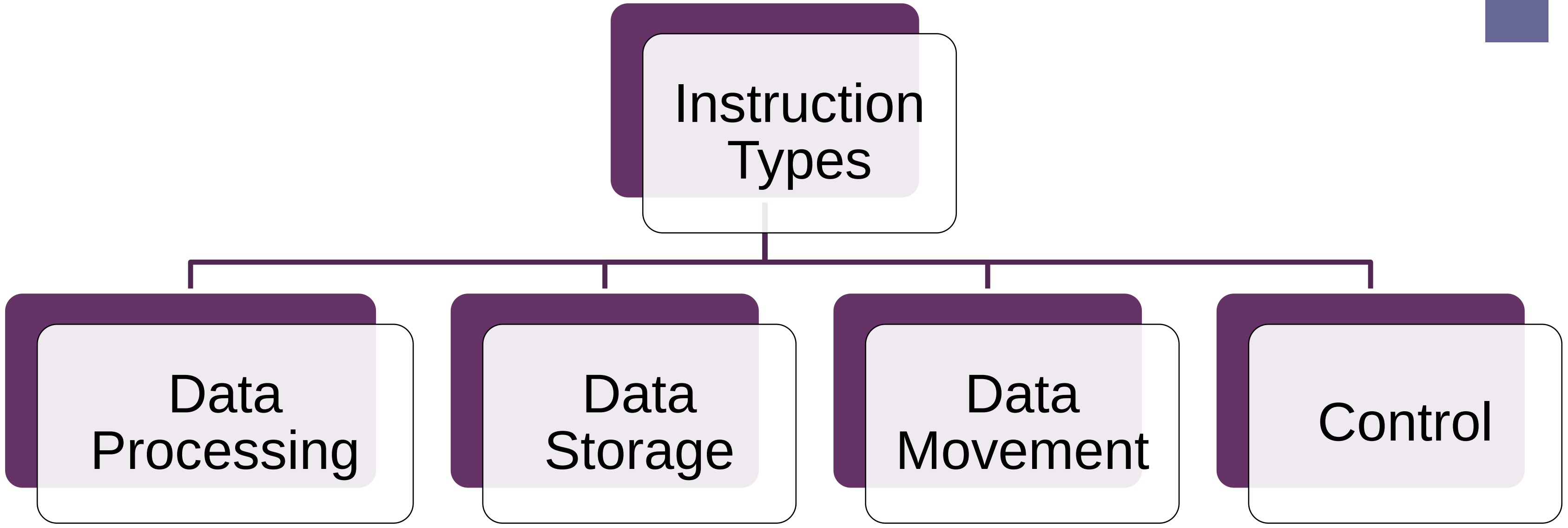
A statement in a high-level language is translated typically into several machine-level instructions

```
mov  eax,Total
add  eax,stuff
mov  Total,eax
```

Intel Assembly Language:

```
A1 00404000
83 00404004
A3 00404008
```

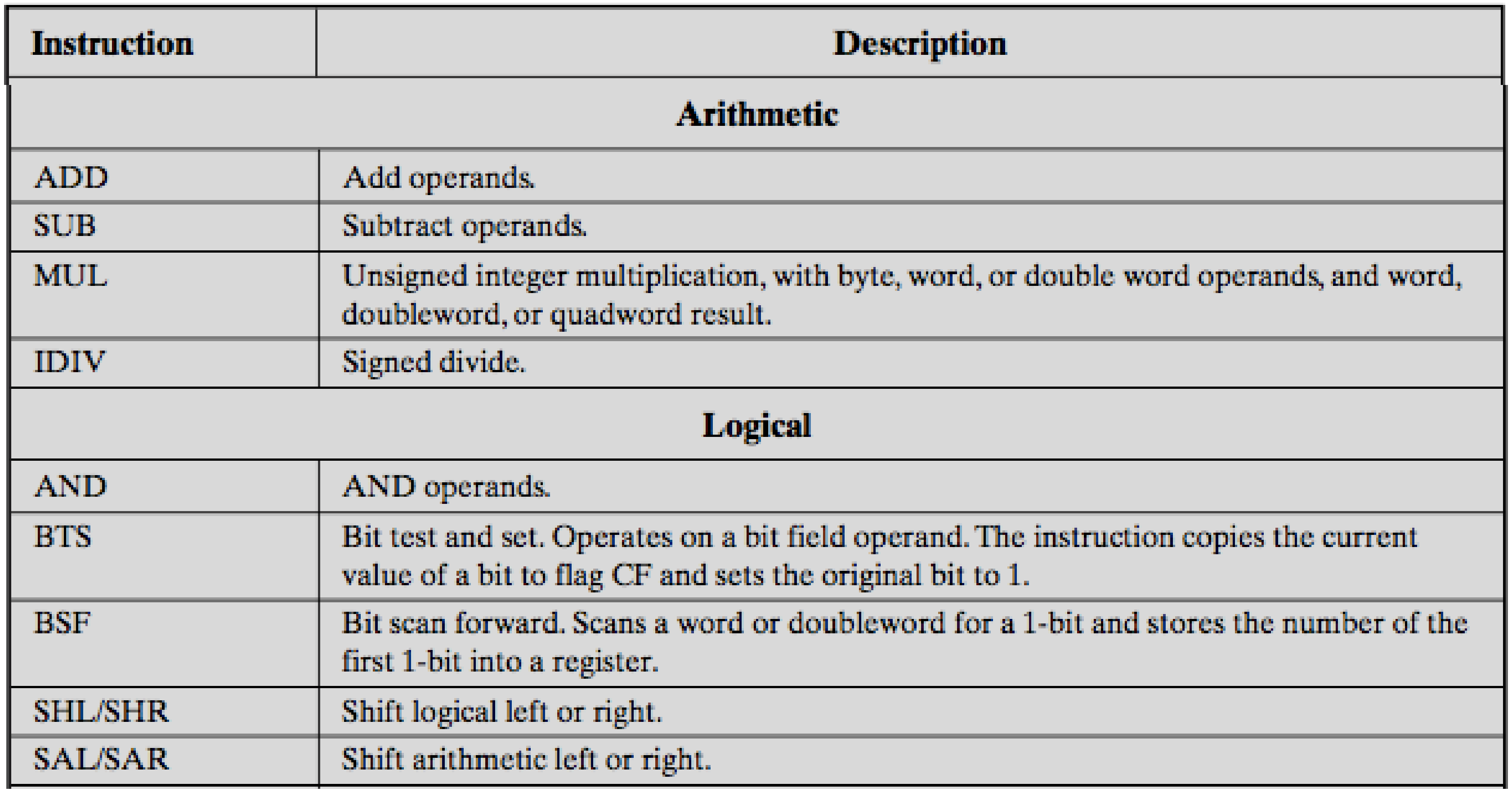
Intel Machine Code:



- *Arithmetic and logic instructions.*
- *Movement of data in/out register and/or memory.*
- *I/O instructions.*
- *Test and branch instructions.*

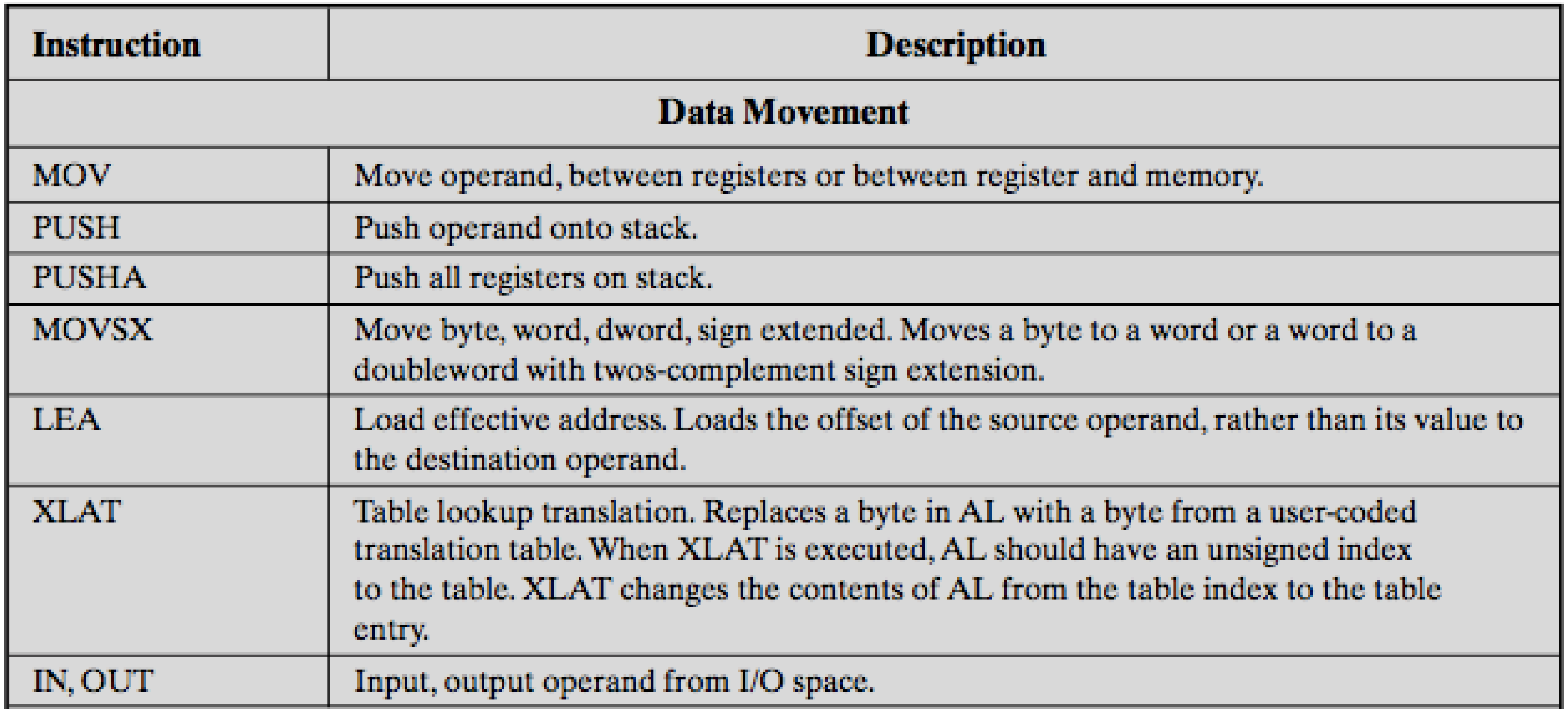
Figure: Categories of instruction types.

Table: Examples of *data processing*.



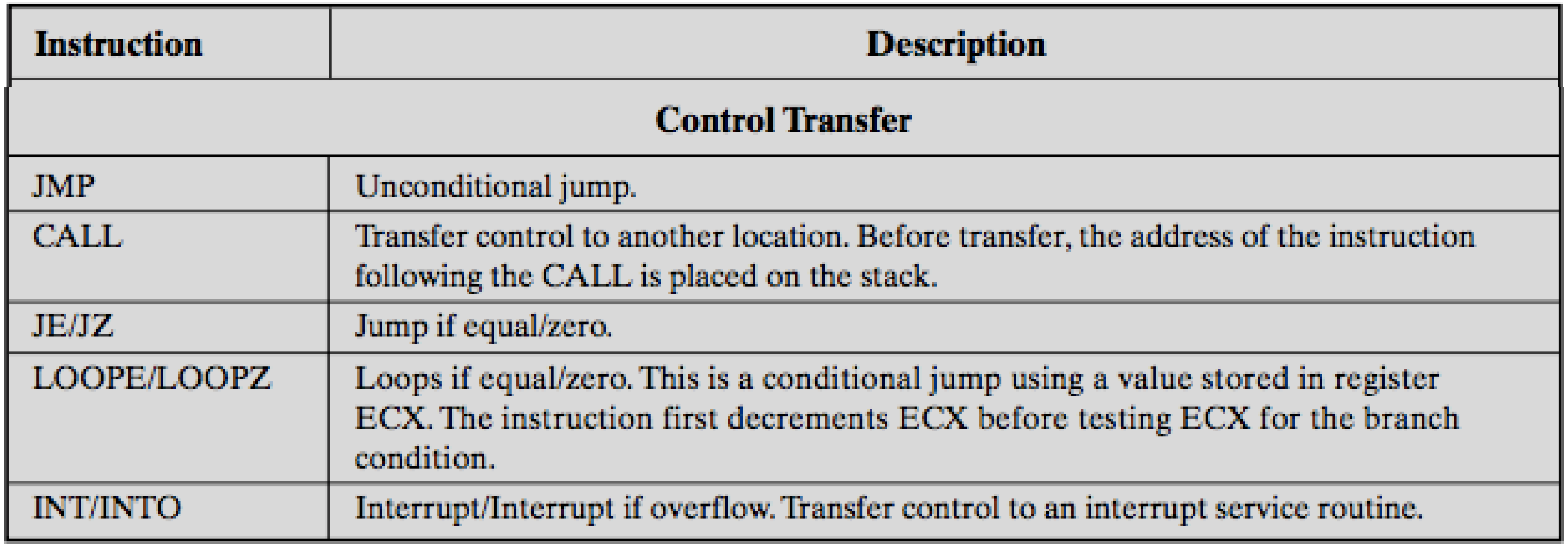
Instruction	Description
Arithmetic	
ADD	Add operands.
SUB	Subtract operands.
MUL	Unsigned integer multiplication, with byte, word, or double word operands, and word, doubleword, or quadword result.
IDIV	Signed divide.
Logical	
AND	AND operands.
BTS	Bit test and set. Operates on a bit field operand. The instruction copies the current value of a bit to flag CF and sets the original bit to 1.
BSF	Bit scan forward. Scans a word or doubleword for a 1-bit and stores the number of the first 1-bit into a register.
SHL/SHR	Shift logical left or right.
SAL/SAR	Shift arithmetic left or right.

Table: Examples of *data storage* and *data movement*.



Instruction	**Description**
	Data Movement
MOV	Move operand, between registers or between register and memory.
PUSH	Push operand onto stack.
PUSHA	Push all registers on stack.
MOVSX	Move byte, word, dword, sign extended. Moves a byte to a word or a word to a doubleword with twos-complement sign extension.
LEA	Load effective address. Loads the offset of the source operand, rather than its value to the destination operand.
XLAT	Table lookup translation. Replaces a byte in AL with a byte from a user-coded translation table. When XLAT is executed, AL should have an unsigned index to the table. XLAT changes the contents of AL from the table index to the table entry.
IN, OUT	Input, output operand from I/O space.

Table: Examples of *control* (test and branch).



Instruction	Description
Control Transfer	
JMP	Unconditional jump.
CALL	Transfer control to another location. Before transfer, the address of the instruction following the CALL is placed on the stack.
JE/JZ	Jump if equal/zero.
LOOPE/LOOPZ	Loops if equal/zero. This is a conditional jump using a value stored in register ECX. The instruction first decrements ECX before testing ECX for the branch condition.
INT/INTO	Interrupt/Interrupt if overflow. Transfer control to an interrupt service routine.

Number of Addresses

- One of the traditional ways in describing processor architecture is using the number of addresses contained in each instruction .

What is the maximum number of addresses one might need in an instruction?

- In most architectures, most instructions have one, two, or three operand addresses.

Example: Program to execute $Y = \dfrac{A - B}{C + (D \times E)}$

Instruction		Comment
SUB	Y, A, B	$Y \leftarrow A - B$
MPY	T, D, E	$T \leftarrow D \times E$
ADD	T, T, C	$T \leftarrow T + C$
DIV	Y, Y, T	$Y \leftarrow Y \div T$

(a) Three-address instructions

4 instructions

Instruction		Comment
MOVE	Y, A	$Y \leftarrow A$
SUB	Y, B	$Y \leftarrow Y - B$
MOVE	T, D	$T \leftarrow D$
MPY	T, E	$T \leftarrow T \times E$
ADD	T, C	$T \leftarrow T + C$
DIV	Y, T	$Y \leftarrow Y \div T$

(b) Two-address instructions

6 instructions

Instruction		Comment
LOAD	D	$AC \leftarrow D$
MPY	E	$AC \leftarrow AC \times E$
ADD	C	$AC \leftarrow AC + C$
STOR	Y	$Y \leftarrow AC$
LOAD	A	$AC \leftarrow A$
SUB	B	$AC \leftarrow AC - B$
DIV	Y	$AC \leftarrow AC \div Y$
STOR	Y	$Y \leftarrow AC$

(c) One-address instructions

8 instructions

PUSH	C
PUSH	D
PUSH	E
MUL	
ADD	
PUSH	B
PUSH	A
SUB	
DIV	
POP	Y

(d) No-address instruction

10 instructions

(a) Three-address instructions

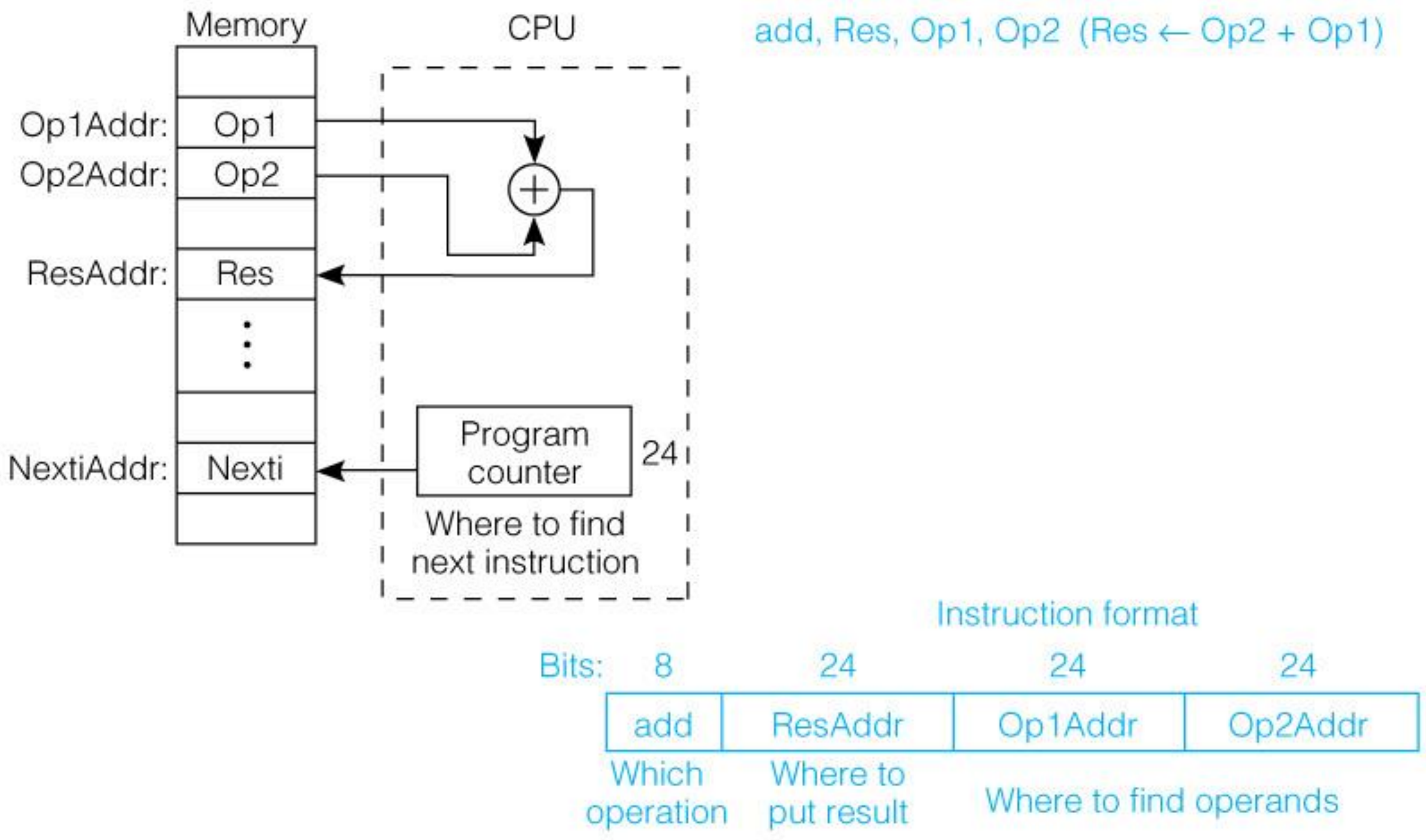
$$Y = \frac{A - B}{C + (D \times E)}$$

Instruction		Comment
SUB	Y, A, B	$Y \leftarrow A - B$
MPY	T, D, E	$T \leftarrow D \times E$
ADD	T, T, C	$T \leftarrow T + C$
DIV	Y, Y, T	$Y \leftarrow Y \div T$

4 instructions

- 3 addresses: *Operand* 1, *Operand* 2, *Result* (Destination).
- May be a forth address - next instruction (usually implicit, obtained from PC (*Program Counter*)).
- Example below: *T* = temporary location used to store intermediate results.
- Not common in use.
- Needs very long words to hold everything.

(a) Three-address instructions



- Address of next instruction kept in a processor state register the - PC (Except for explicit Branches/Jumps)

(b) Two-address instructions

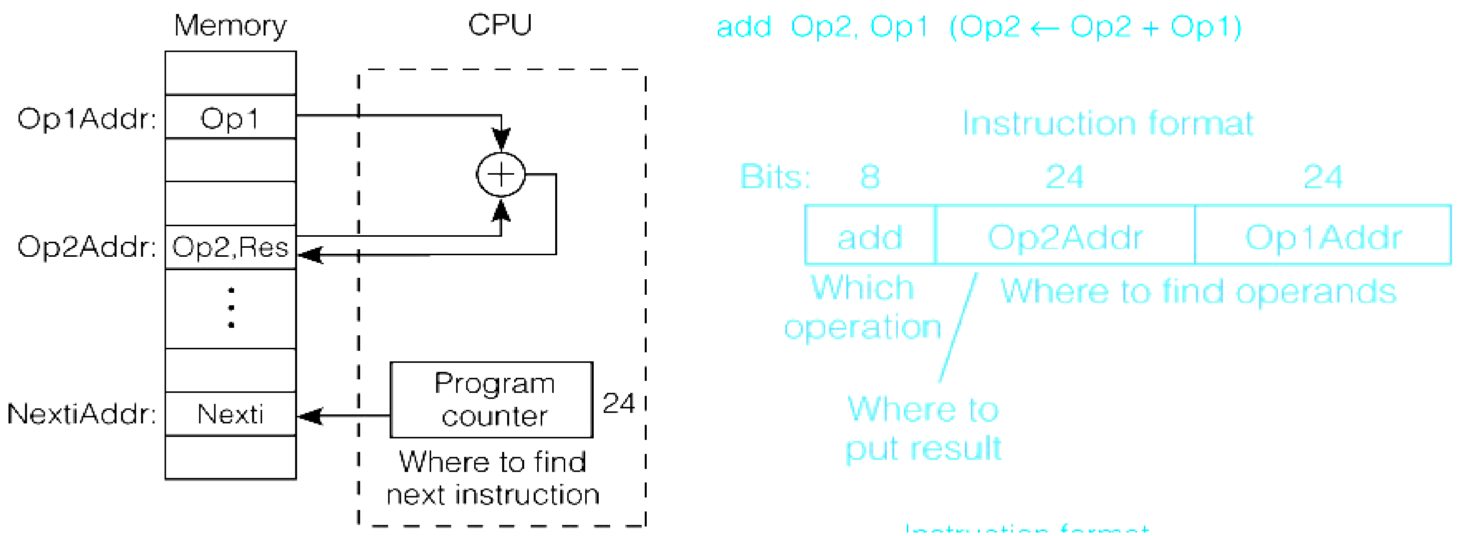
$$Y = \frac{A - B}{C + (D \times E)}$$

Instruction		Comment
MOVE	Y, A	$Y \leftarrow A$
SUB	Y, B	$Y \leftarrow Y - B$
MOVE	T, D	$T \leftarrow D$
MPY	T, E	$T \leftarrow T \times E$
ADD	T, C	$T \leftarrow T + C$
DIV	Y, T	$Y \leftarrow Y \div T$

6 instructions

- 2 addresses: *Operand*, *Result* (Destination).
- Reduces length of instruction and space requirements.
- Requires some extra works:
  - Temporary storage to hold some results.
  - Done to avoid altering the operand value.

(b) Two-address instructions



- Be aware of the difference between address, *Op1Addr*, and data stored at that address, *Op1*.
- Result overwrites operand 2, *Op2*, with result, *Res*
- This format needs only 2 addresses in the instruction but there is less choice in placing data

(c) One-address instructions

$$Y = \frac{A - B}{C + (D \times E)}$$

Instruction	Comment
LOAD D	$AC \leftarrow D$
MPY E	$AC \leftarrow AC \times E$
ADD C	$AC \leftarrow AC + C$
STOR Y	$Y \leftarrow AC$
LOAD A	$AC \leftarrow A$
SUB B	$AC \leftarrow AC - B$
DIV Y	$AC \leftarrow AC \div Y$
STOR Y	$Y \leftarrow AC$

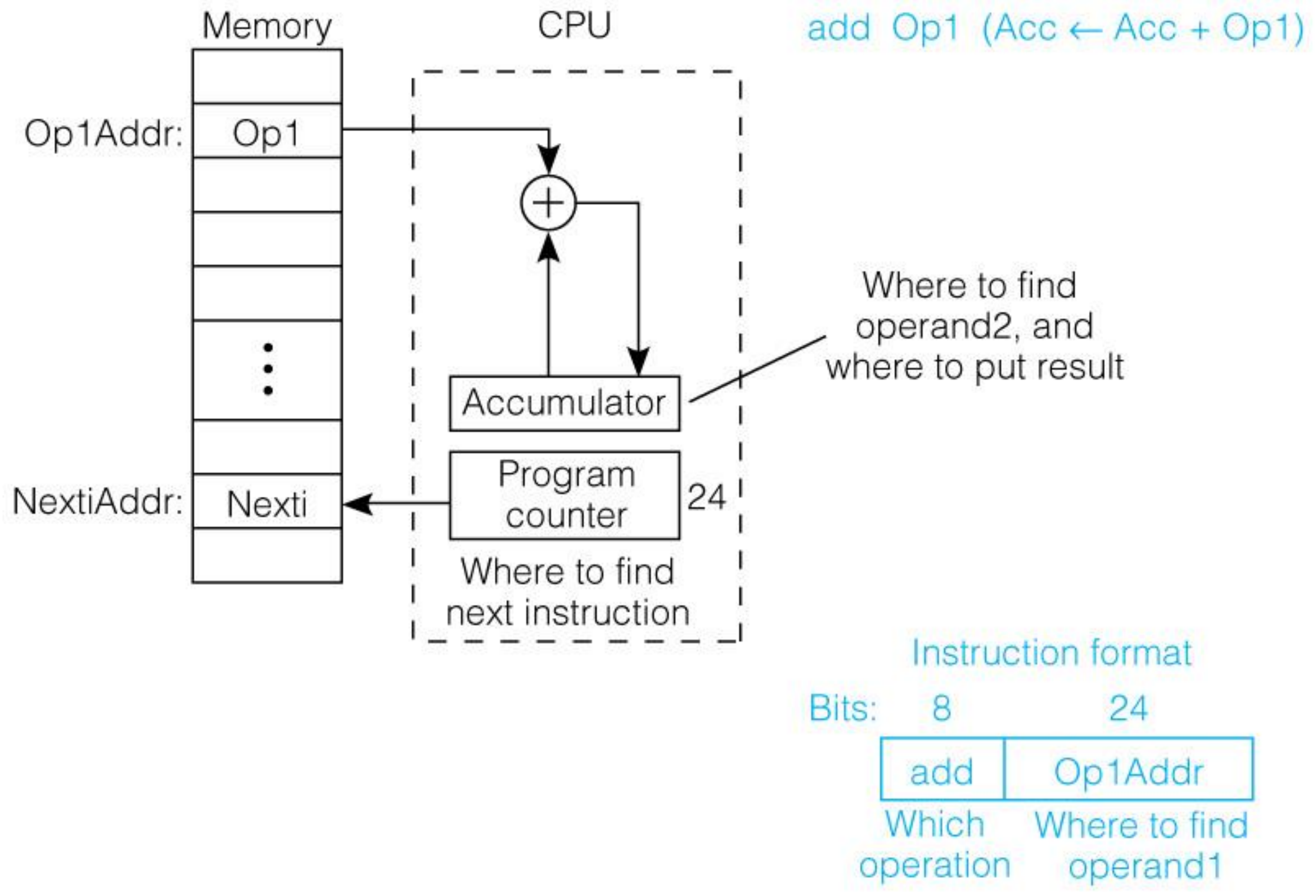
8 instructions

- 1 address
- Implicit second address.
- Usually use a *CPU register* (accumulator)
  - Supplies 1 operand and store result.
  - One memory address used for other operand.
- Common on early machines.

(c) One-address instructions

We now need instructions to load and store operands:

LDA OpAddr
STA OpAddr



- Special CPU register, the accumulator, supplies 1 operand and stores result
- One memory address used for other operand

(d) No-address instructions

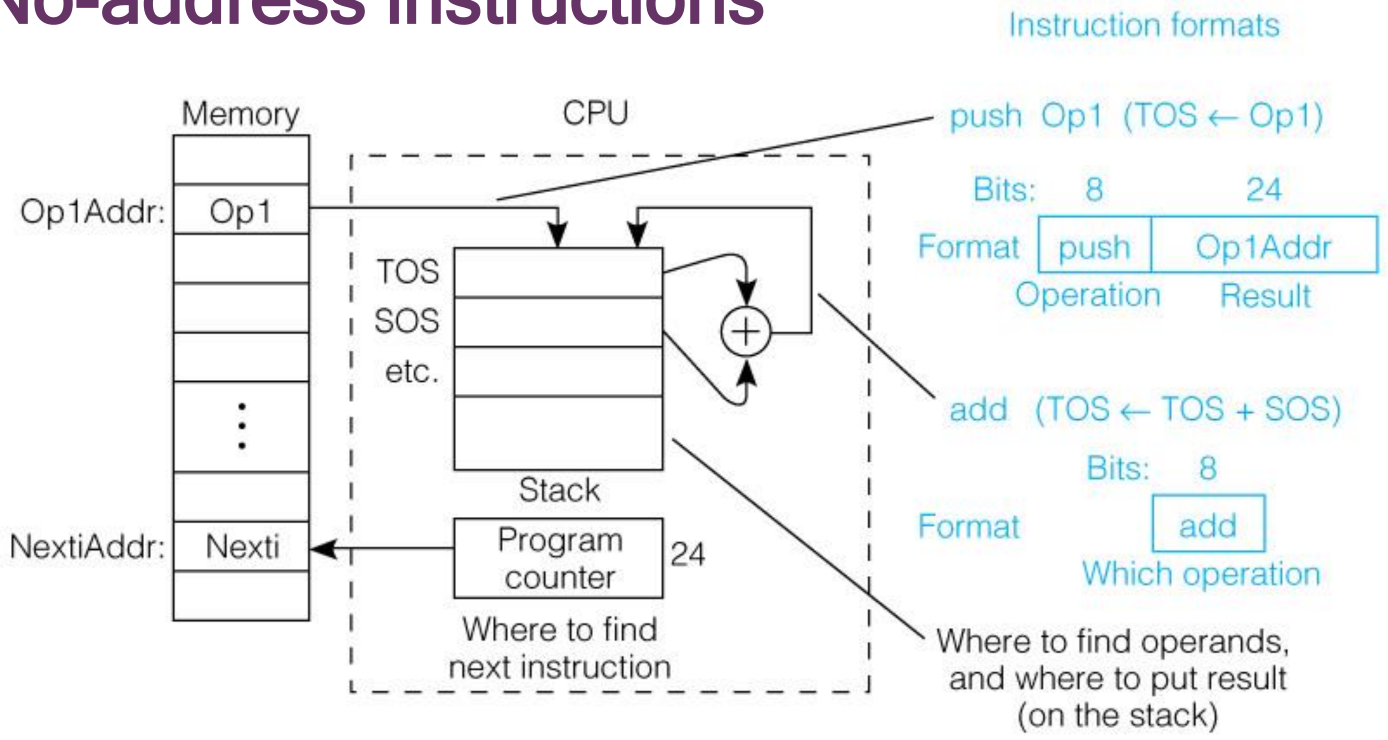
$$Y = \frac{A - B}{C + (D \times E)}$$

```
PUSH   C
PUSH   D
PUSH   E
MUL
ADD
PUSH   B
PUSH   A
SUB
DIV
POP    Y
```

10 instructions

- 0 (zero) address
- All addresses implicit.
- Usually use a *stack* (a push down stack in CPU).
- There are two *Opcodes* with one *operand*: PUSH op, POP op.

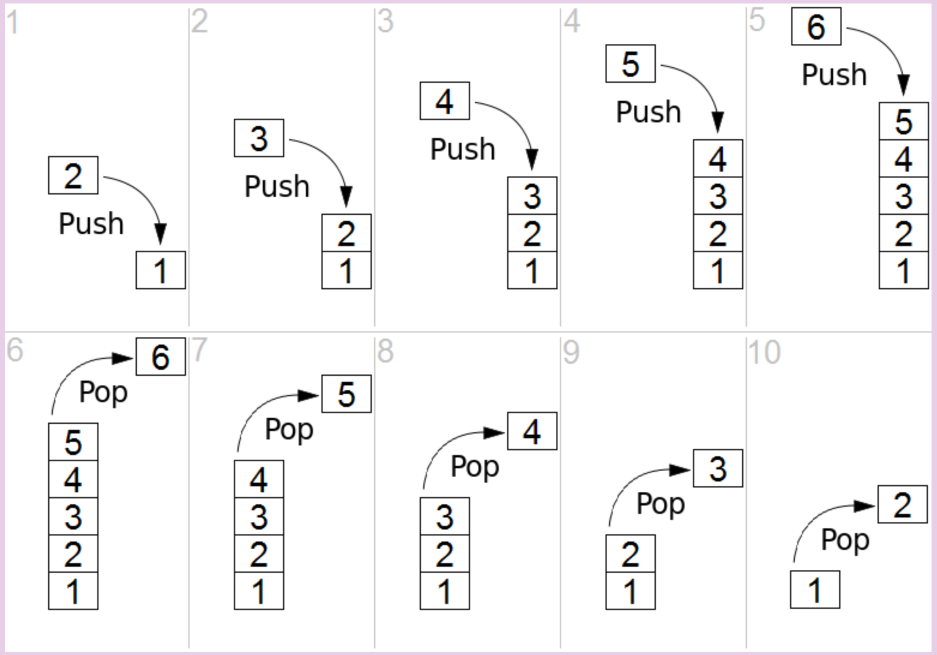
(d) No-address instructions



- Uses a push down stack in CPU
- Arithmetic uses stack for both operands. The result replaces them on the TOS
- Computer must have a 1 address instruction to push and pop operands to and from the stack

(d) No-address instructions

Stack Machine

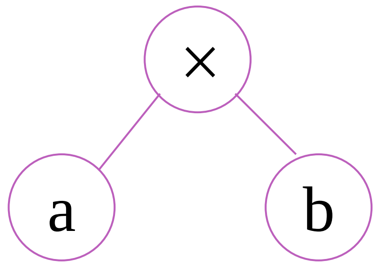


- A *stack* is an abstract data type and data structure based on the principle of *Last In First Out* (LIFO).
- *Stack machine*: Java Virtual Machine.
- *Call stack* of a program, also known as a function stack, execution stack, control stack, or simply the stack.
- Application: *Reverse Polish Notation (RPN)*, *Depth-First-Search (DFS)*

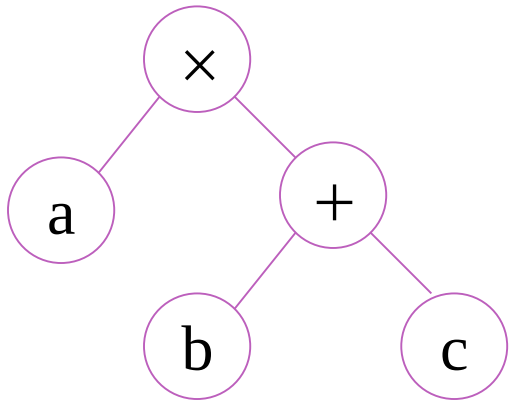
RPN will be discussed in next example

Expression tree

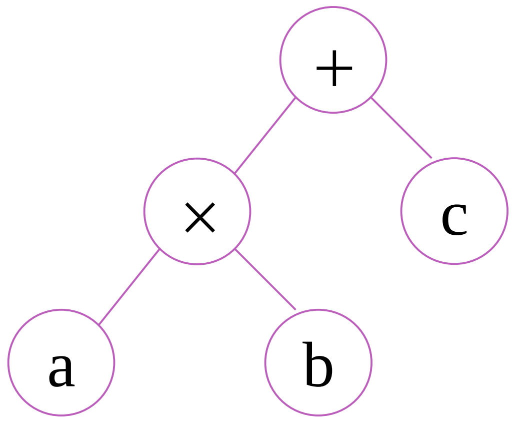
$a \times b$



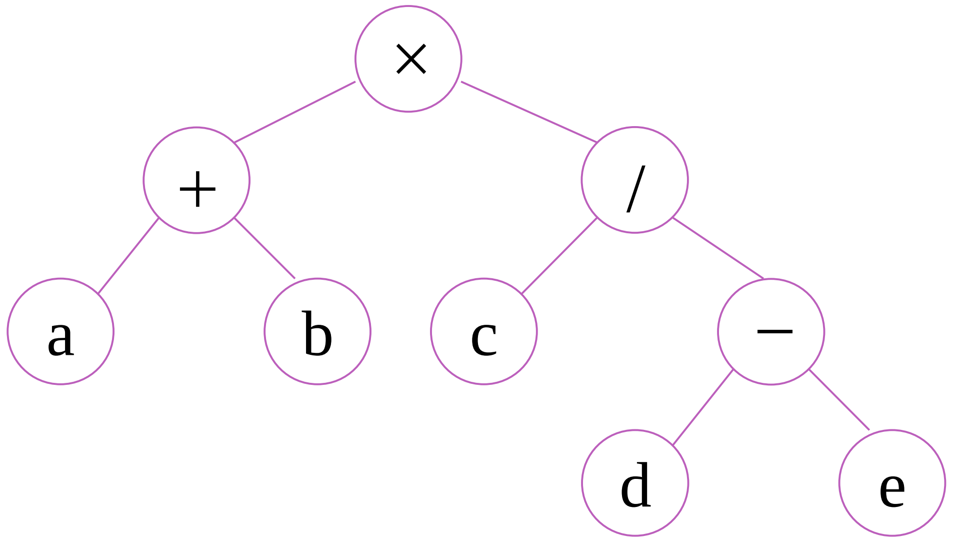
$a \times (b + c)$



$a \times b + c$



$(a + b) \times (c / (d - e))$

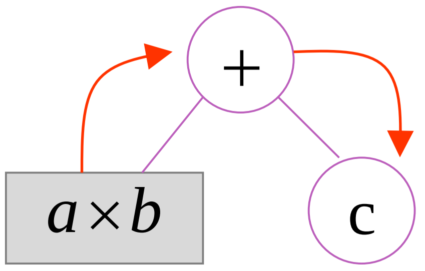


Expression tree:

(a) Infix notation:
Left-Parent-Right order

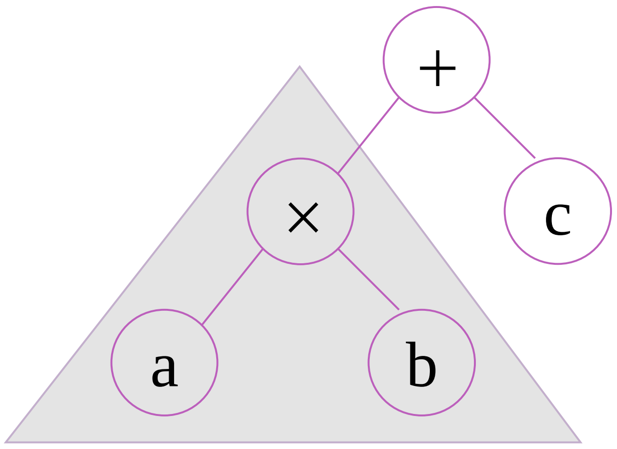
Recursive
Left-Parent-
Right again

infix : $(a \times b) + c$



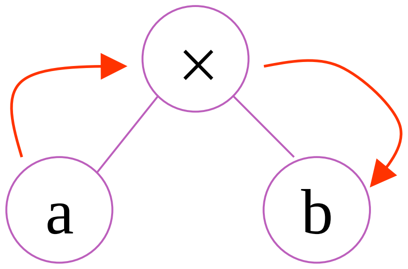
Replace
subtree with
infix notation

$a \times b + c$



Left child of root "+"

Recursive
Left-Parent-Right



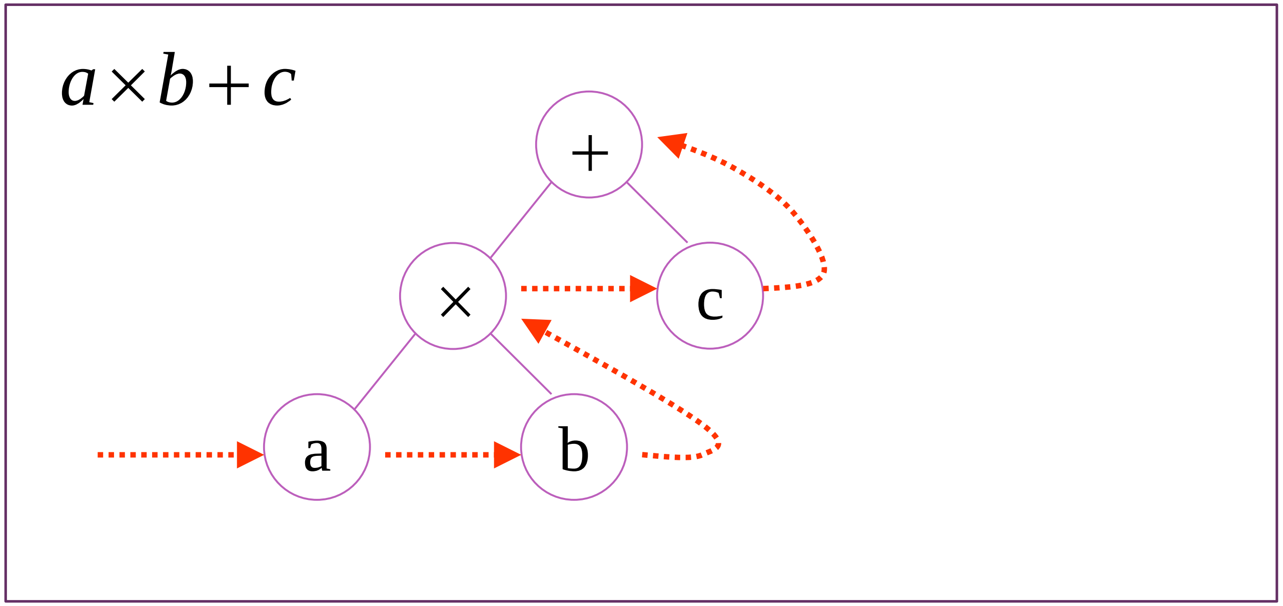
infix : $a \times b$

Expression tree:

(b) Postfix notation:

Left-Right-Parent order

$$postfix: ab\times c+$$



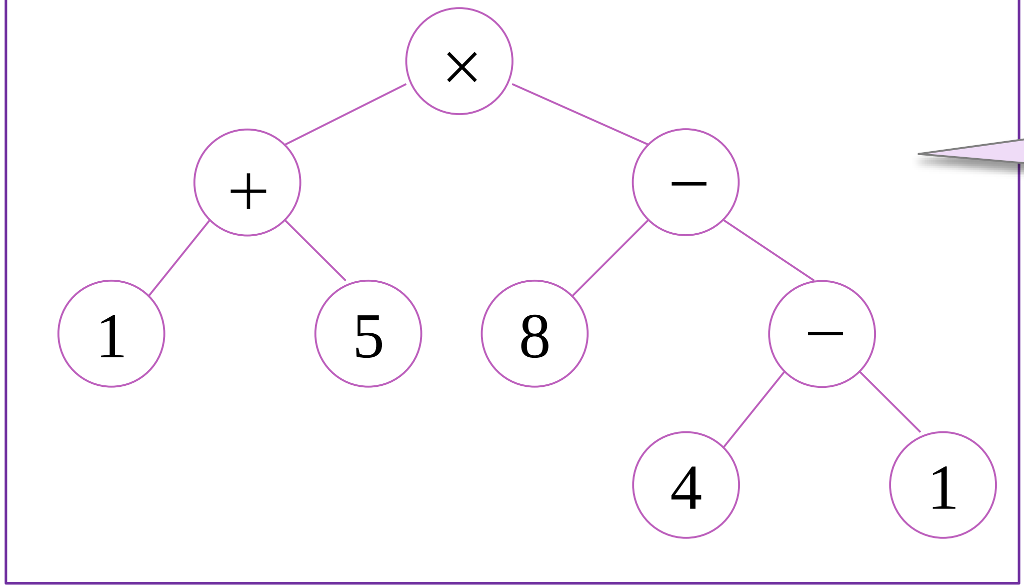
Reverse Polish Notation (RPN)

- Precedence of multiplication is higher than addition, we need parenthesis to guarantee execution order.
- However in the early 1950s, the Polish logician Jan Lukasiewicz observed that parentheses are not necessary in postfix notation, called *RPN* (*Reverse Polish Notation*).
- The Reverse Polish scheme was proposed by F.L. Bauer and E.W. Dijkstra in the early 1960s to reduce computer memory access and utilize the *stack* to evaluate expressions .

Example: Reverse Polish Notation (RPN) → *Postfix order*

$$\textit{Infix}: (1+5)\times(8-(4-1))$$

Expression tree



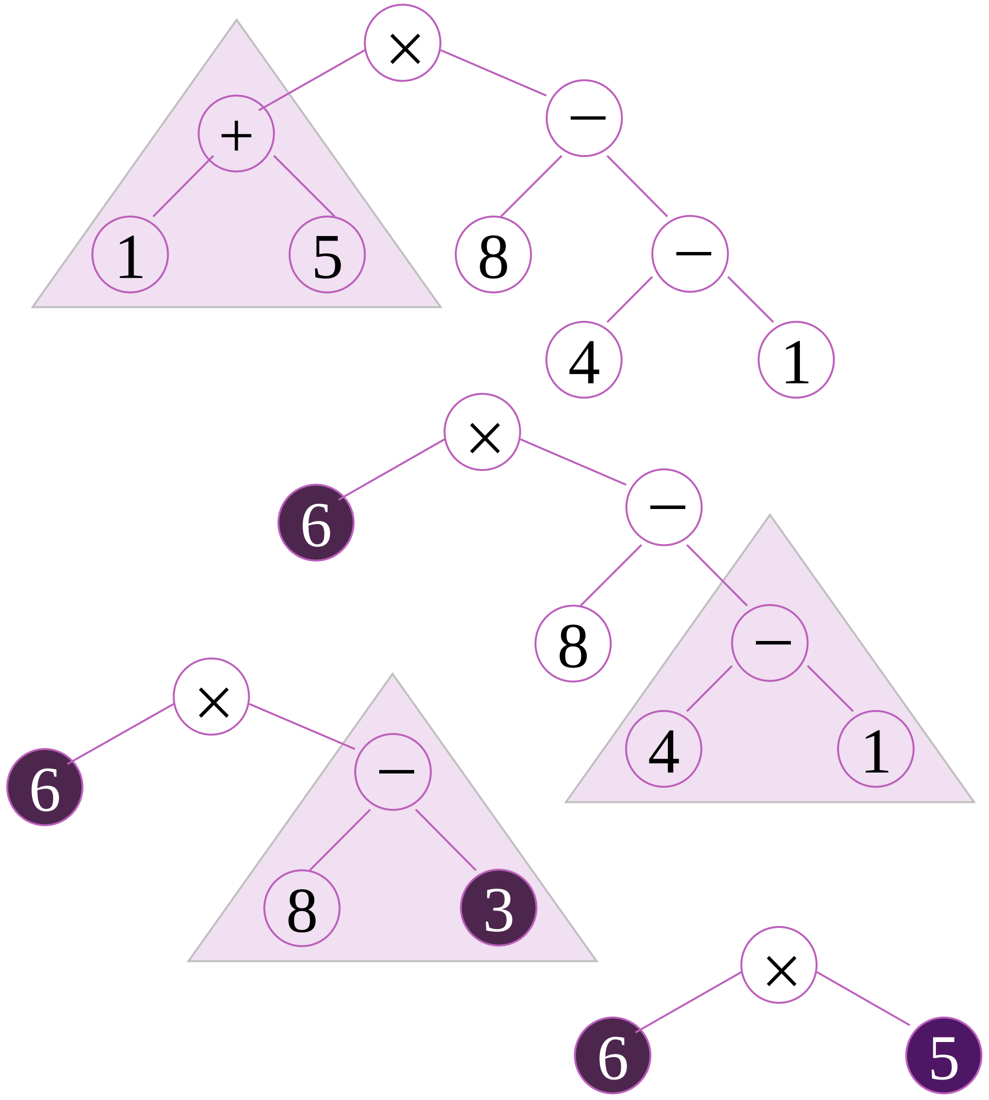
Postfix:

$$1\,5 + 8\,4\,1 - - \times$$

(parenthesis free)

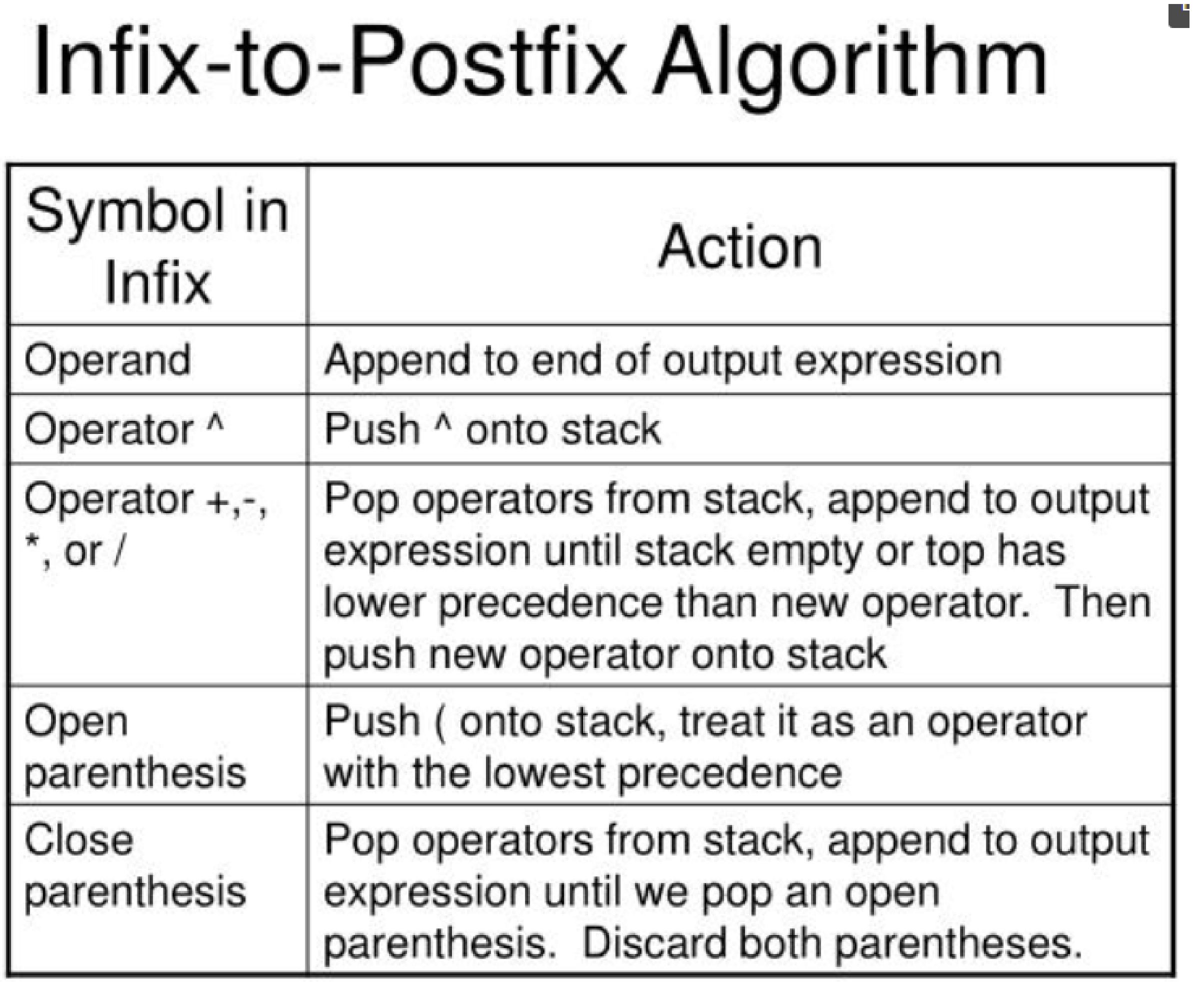
Example: Reverse Polish Notation (RPN) → *Postfix order* [1]

$$\mathit{Infix}: (1+5)\times(8-(4-1))$$



$$\begin{tikzcd}[column sep=small]
\mathit{Postfix}: \underline{1\,5\,+}\,8\,4\,1\,-\,-\,\times \arrow[d, "1+5=6"] \\
6\,8\,\underline{4\,1\,-}\,-\,\times \arrow[d, "4-1=3"] \\
6\,\underline{8\,3\,-}\,\times \arrow[d, "8-3=5"] \\
\underline{6\,5\,\times} \arrow[d, "6\times5=30"] \\
30
\end{tikzcd}$$

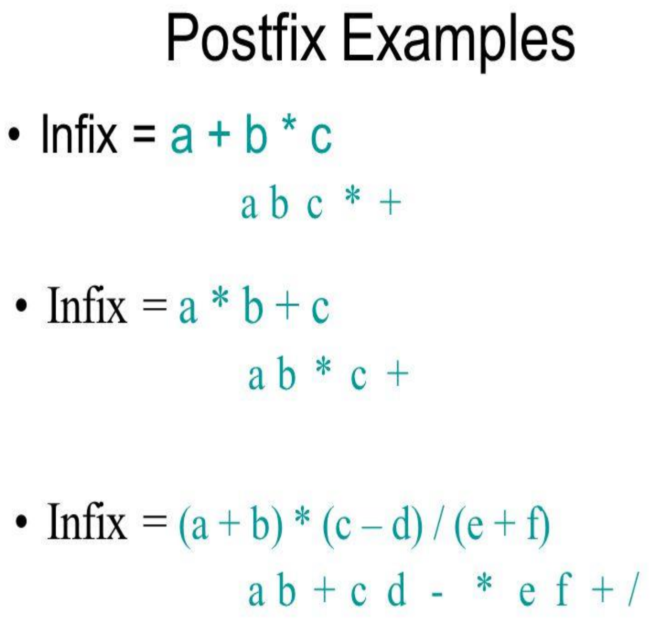
Converting Infix to Postfix Algorithm



Infix-to-Postfix Algorithm

Symbol in Infix	Action
Operand	Append to end of output expression
Operator ^	Push ^ onto stack
Operator +,-, *, or /	Pop operators from stack, append to output expression until stack empty or top has lower precedence than new operator. Then push new operator onto stack
Open parenthesis	Push (onto stack, treat it as an operator with the lowest precedence
Close parenthesis	Pop operators from stack, append to output expression until we pop an open parenthesis. Discard both parentheses.

Converting Infix to Postfix Algorithm



Postfix Examples

- Infix = a + b * c

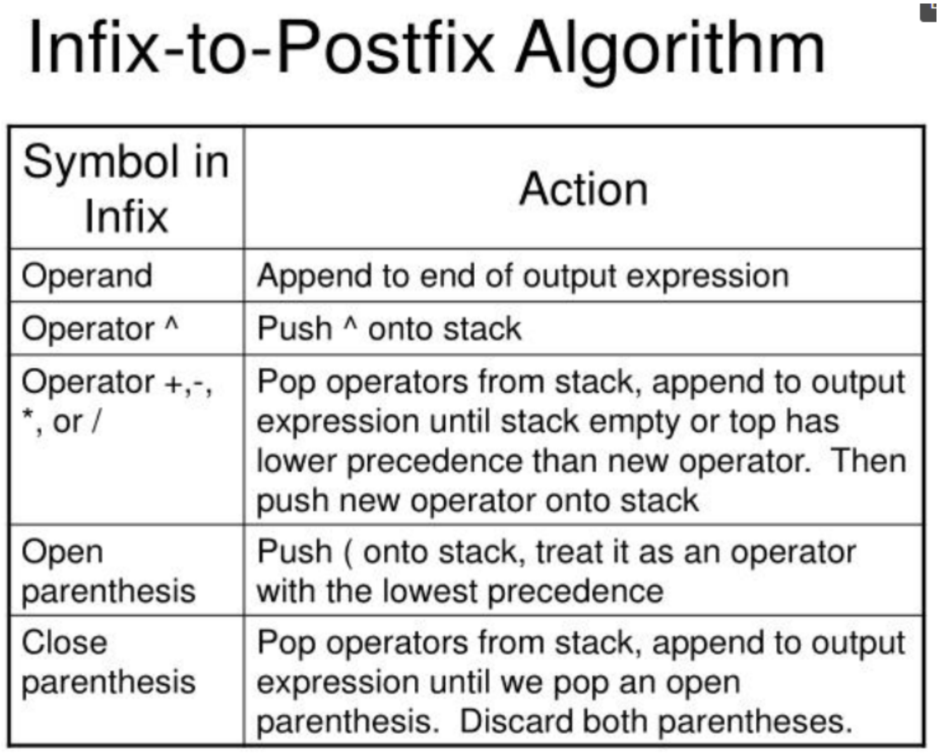
  a b c * +

- Infix = a * b + c

  a b * c +

- Infix = (a + b) * (c – d) / (e + f)

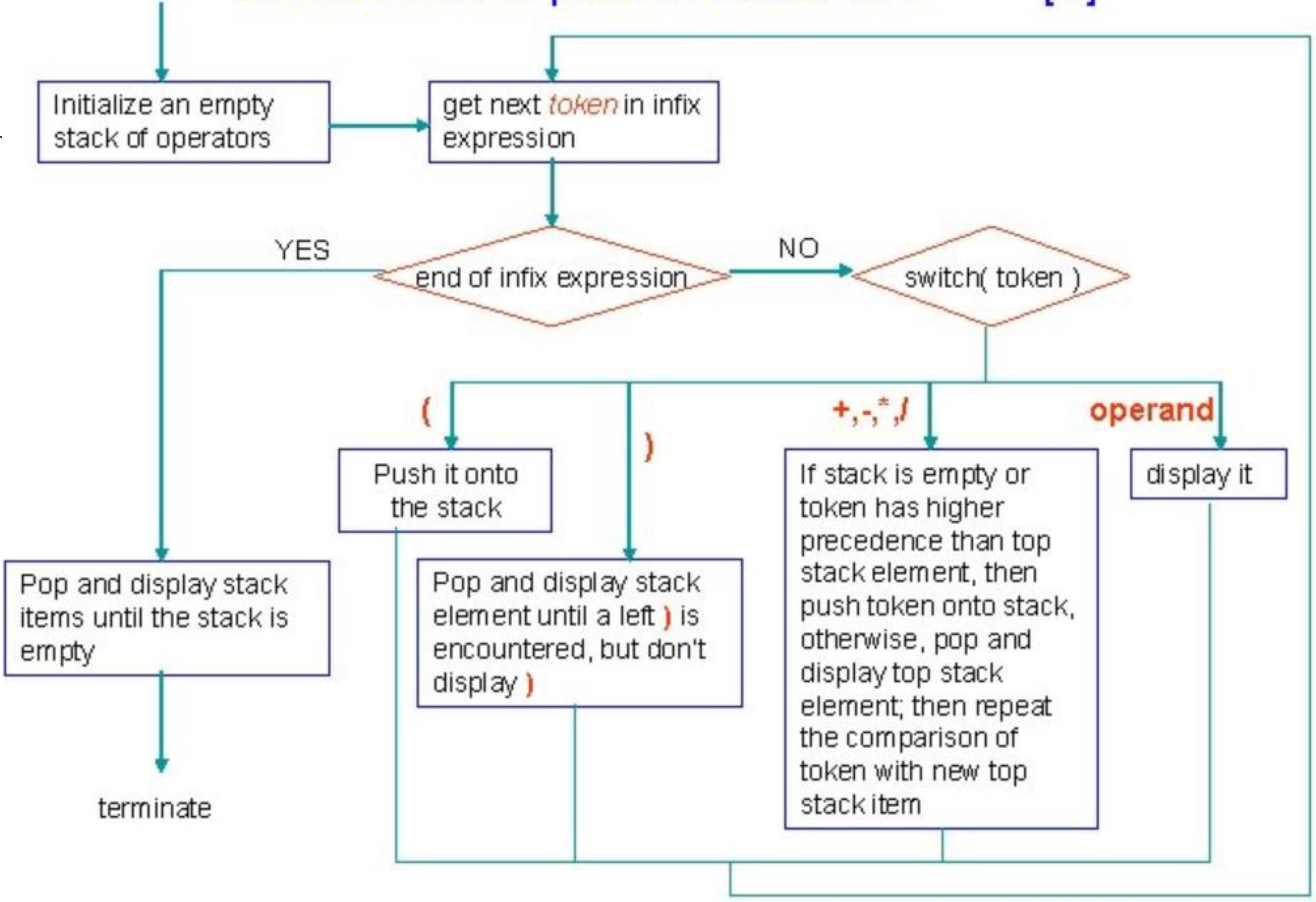
  a b + c d - * e f + /



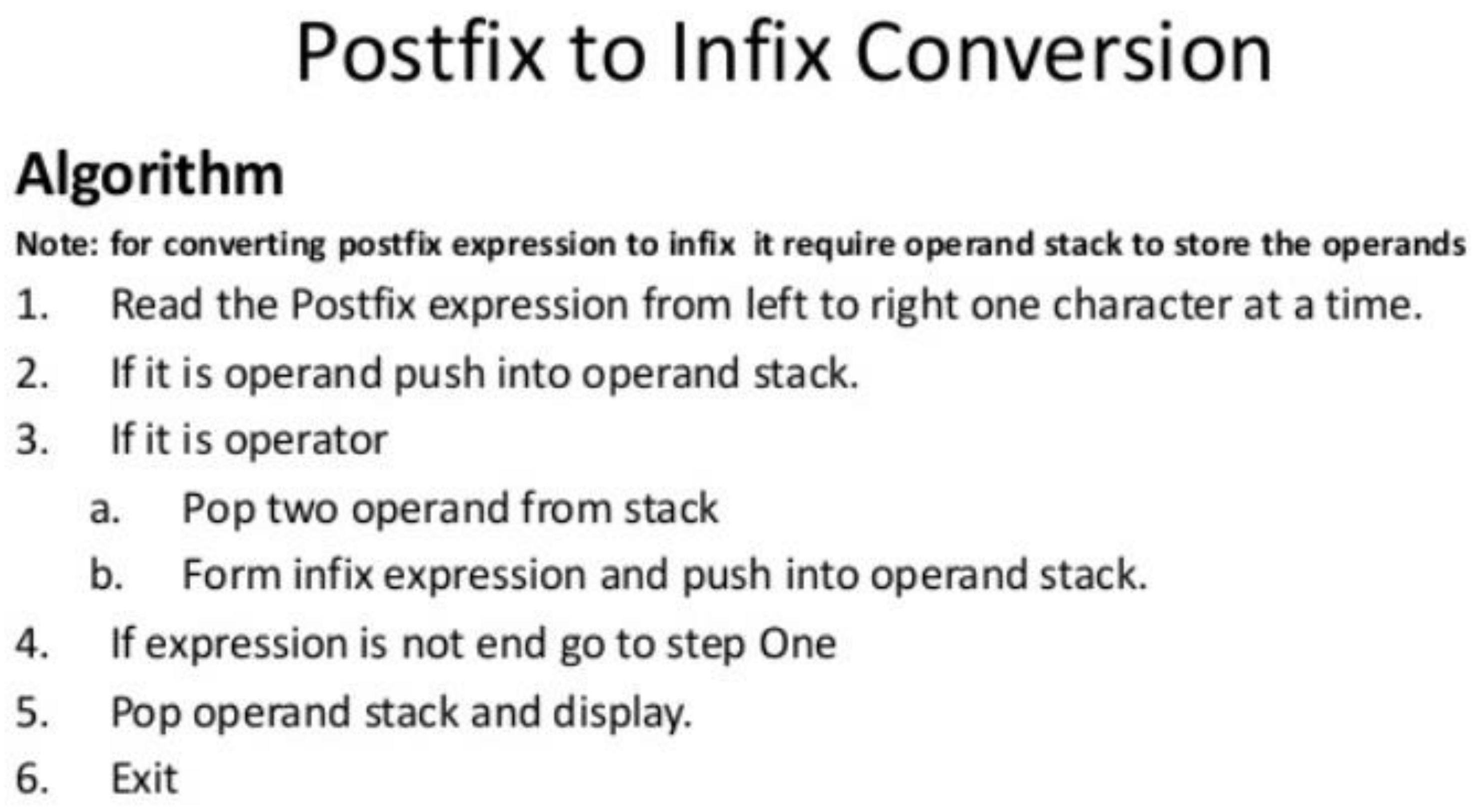
Infix-to-Postfix Algorithm

Symbol in Infix	Action
Operand	Append to end of output expression
Operator ^	Push ^ onto stack
Operator +,-, *, or /	Pop operators from stack, append to output expression until stack empty or top has lower precedence than new operator. Then push new operator onto stack
Open parenthesis	Push (onto stack, treat it as an operator with the lowest precedence
Close parenthesis	Pop operators from stack, append to output expression until we pop an open parenthesis. Discard both parentheses.

convert infix to postfix: flow chart [5]


Initialize an empty stack of operators
get next token in infix expression
end of infix expression
YES
NO
switch(token)
(
)
+,-,*,/
operand
Push it onto the stack
Pop and display stack element until a left) is encountered, but don't display)
If stack is empty or token has higher precedence than top stack element, then push token onto stack, otherwise, pop and display top stack element; then repeat the comparison of token with new top stack item
display it
Pop and display stack items until the stack is empty
terminate

Converting Postfix to Infix Algorithm



Postfix to Infix Conversion

Algorithm

Note: for converting postfix expression to infix it require operand stack to store the operands

1. Read the Postfix expression from left to right one character at a time.
2. If it is operand push into operand stack.
3. If it is operator
    a. Pop two operand from stack
    b. Form infix expression and push into operand stack.
4. If expression is not end go to step One
5. Pop operand stack and display.
6. Exit

Example: Evaluate RPN expression [1]

Expression

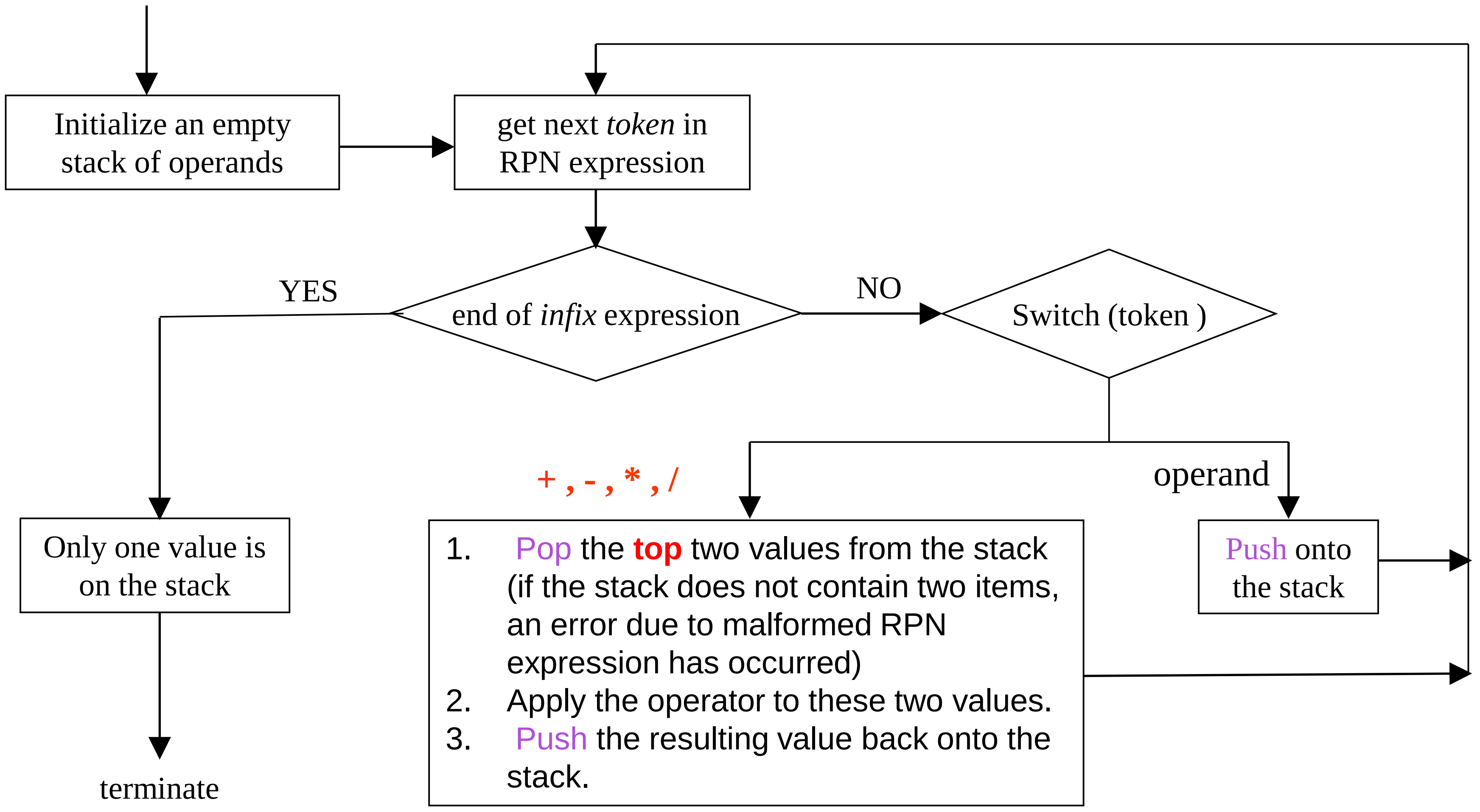
$\boxed{30}$

Infix : $(1+5)\times(8-(4-1))$

Postfix : $1\,5+8\,4\,1--\times$

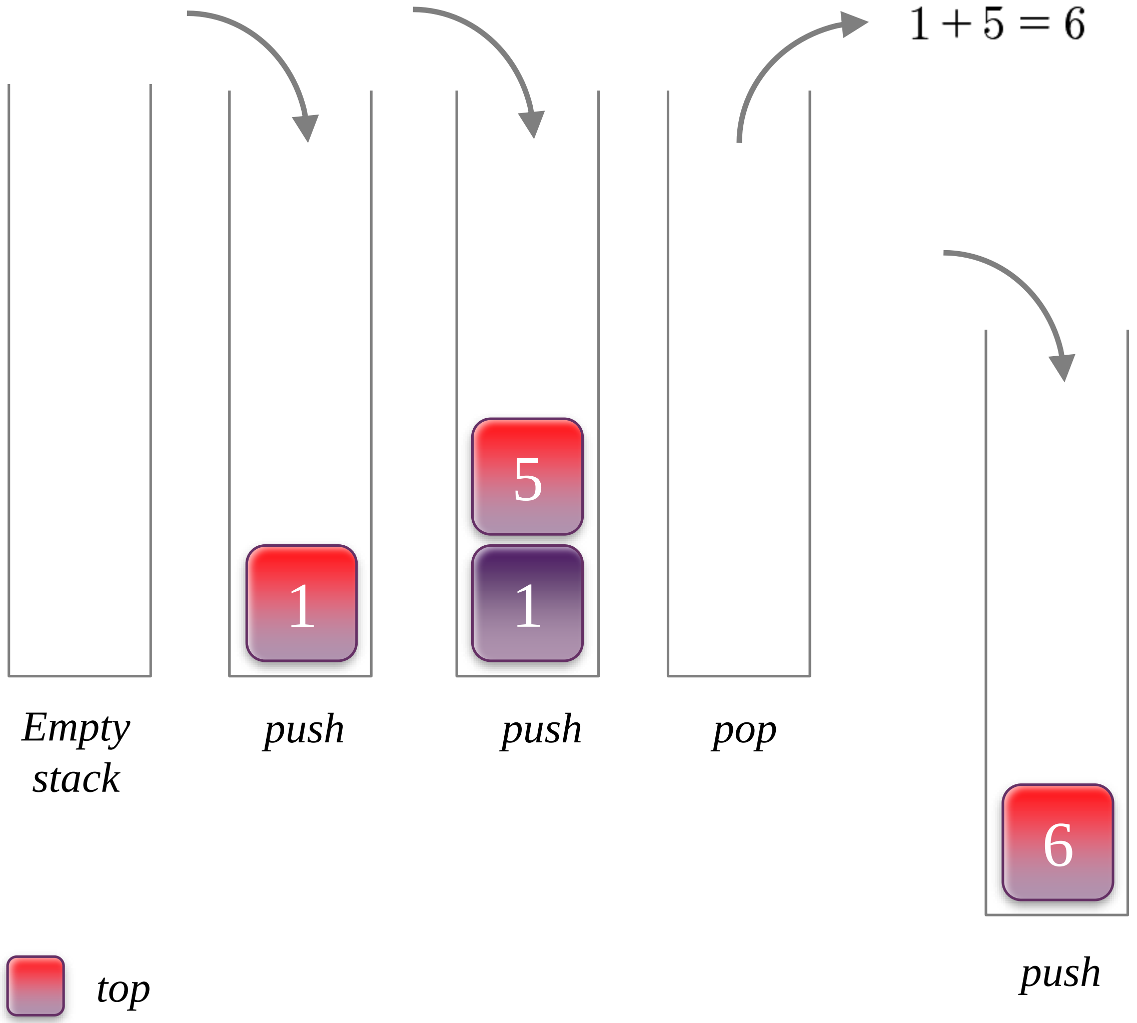
- **Scanned** from left to right until an operator is found, then the last two operands must be retrieved and combined.
- Order of operands satisfy LIFO, so we can use **stack** to store operands and then evaluate RPN expression.

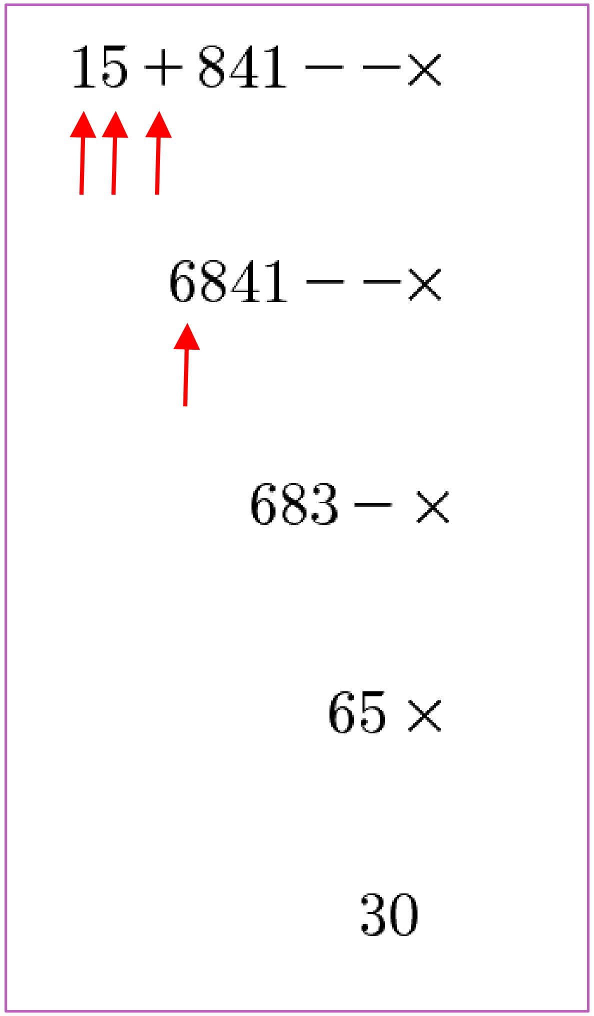
Example: Evaluate RPN expression → *Flow Chart*


Initialize an empty stack of operands
get next *token* in RPN expression
end of *infix* expression
YES
NO
Switch (token)
+ , - , * , /
operand
Only one value is on the stack
terminate
Push onto the stack

1. Pop the **top** two values from the stack (if the stack does not contain two items, an error due to malformed RPN expression has occurred)
2. Apply the operator to these two values.
3. Push the resulting value back onto the stack.

Example: Evaluate RPN expression [2]




$$15+841--\times$$

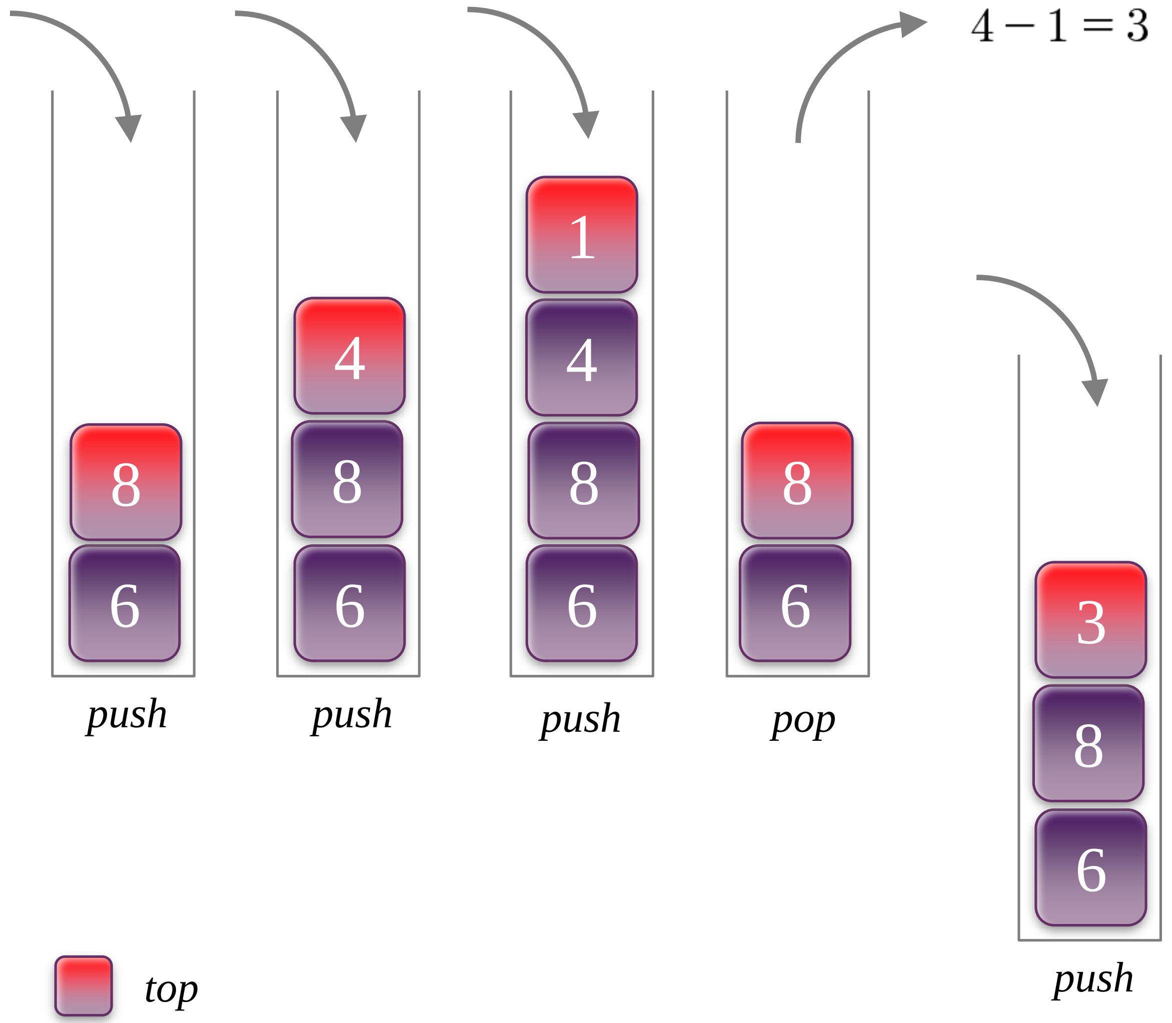
$$6841--\times$$

$$683-\times$$

$$65\times$$

$$30$$

Example: Evaluate RPN expression [2]



$$15+841--\times$$

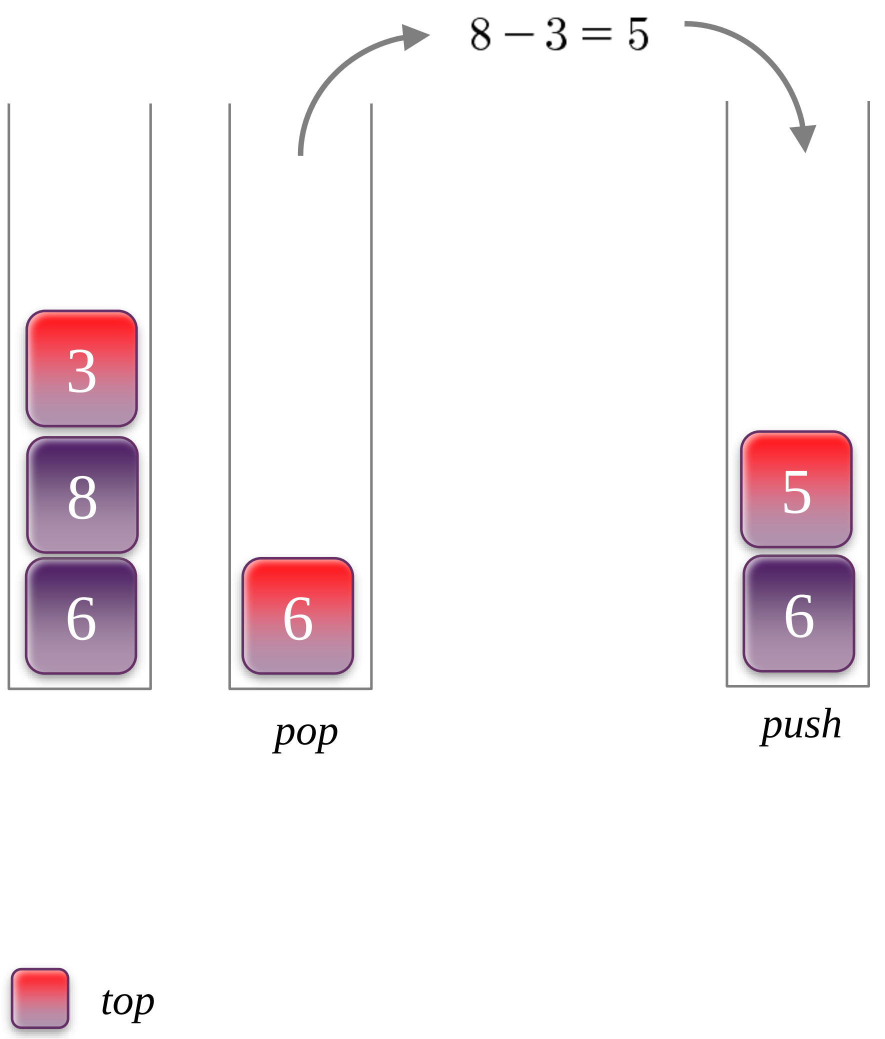
$$6841--\times$$

$$683-\times$$

$$65\times$$

$$30$$

Example: Evaluate RPN expression [2]



$$15 + 8 4 1 - - \times$$

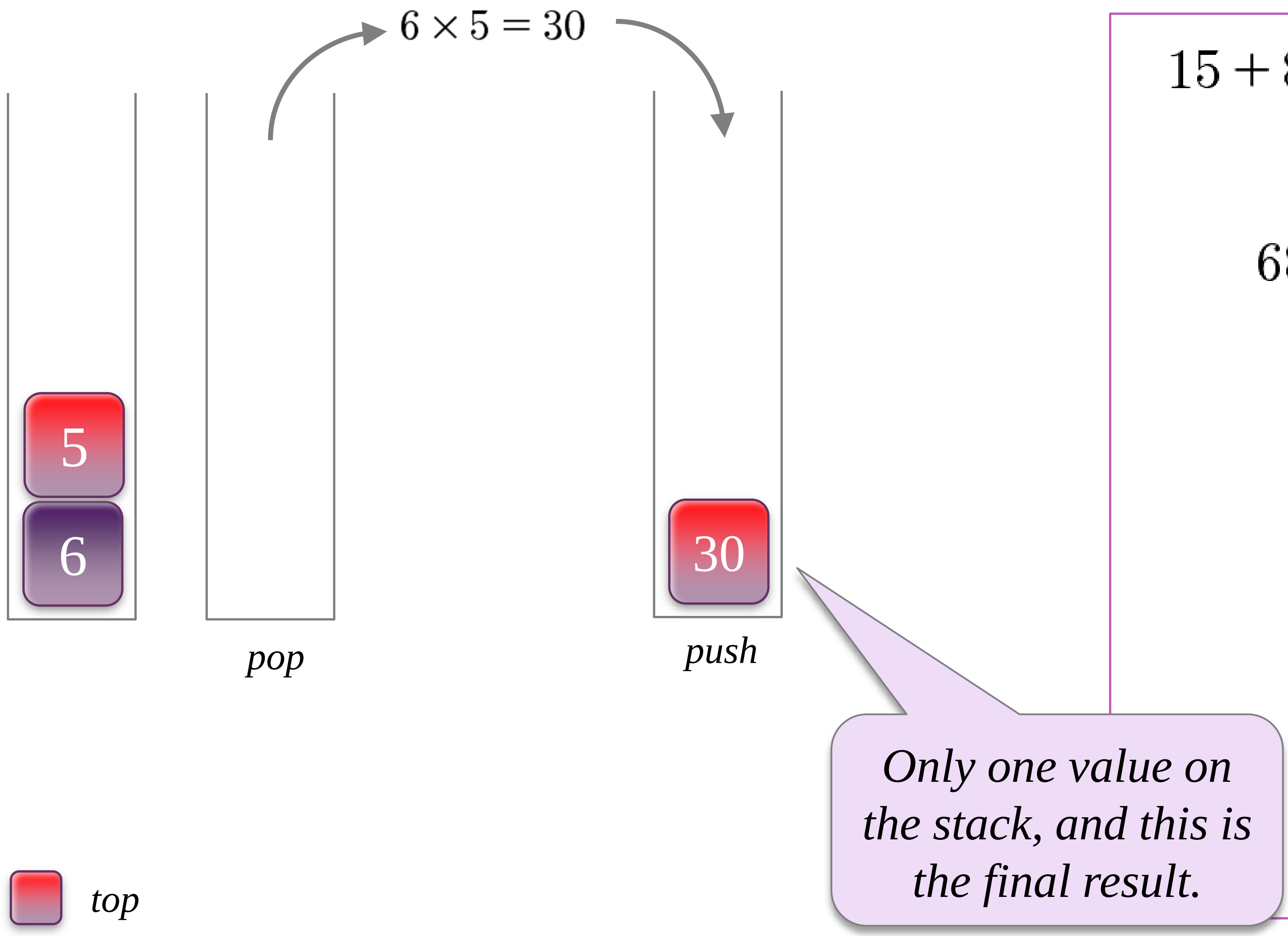
$$6 8 4 1 - - \times$$

$$6 8 3 - \times$$

$$6 5 \times$$

$$30$$

Example: Evaluate RPN expression [2]



$$15 + 8 4 1 - - \times$$

$$6 8 4 1 - - \times$$

$$6 8 3 - \times$$

$$6 5 \times$$

$$30$$

Example

Exercise 4.1a:

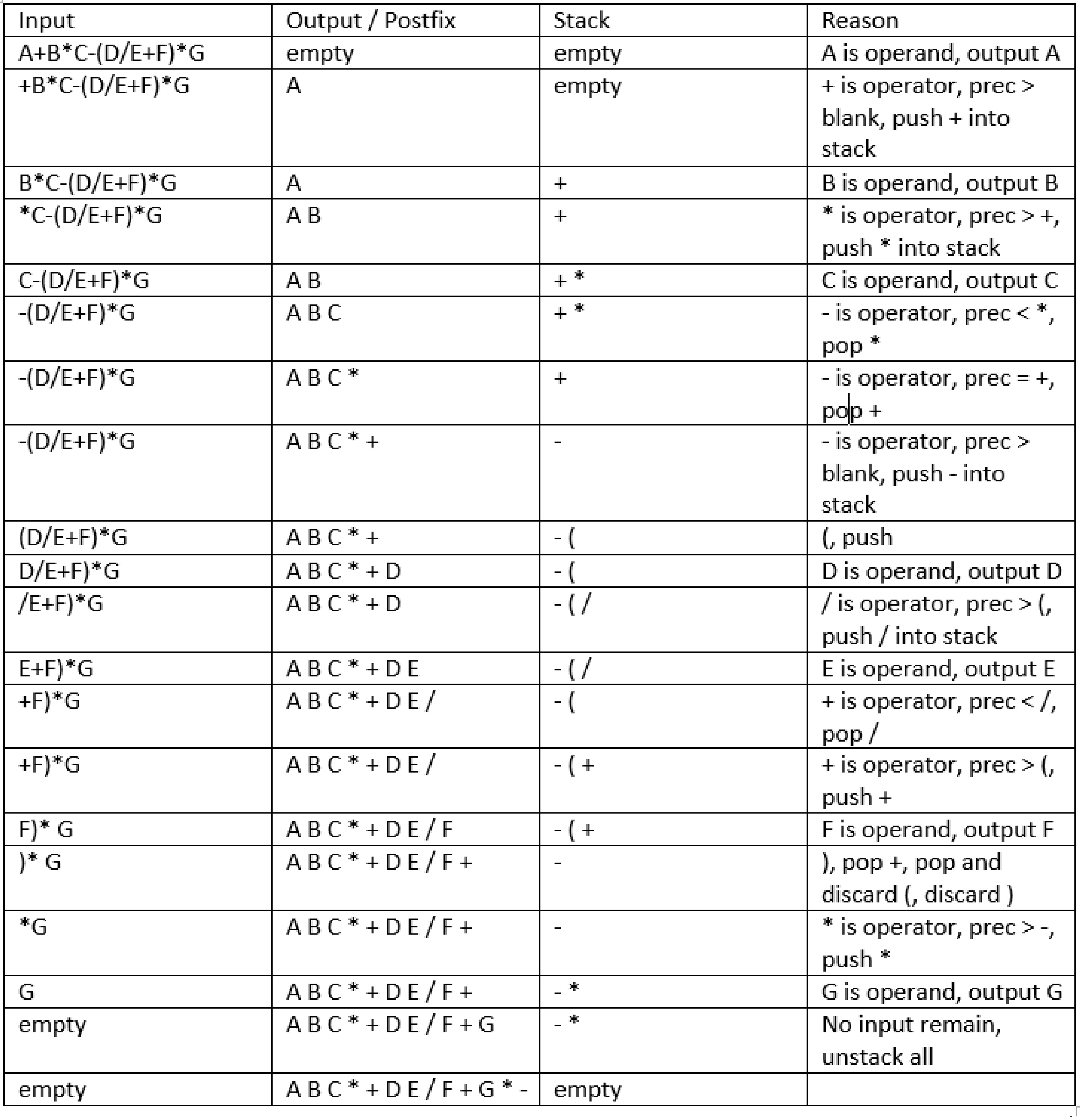
Given an expressi

$A+B*C-(D/E$

(a) Construct the e

(b) Convert into Pl

evaluation.



Input	Output / Postfix	Stack	Reason
A+B*C-(D/E+F)*G	empty	empty	A is operand, output A
+B*C-(D/E+F)*G	A	empty	+ is operator, prec > blank, push + into stack
B*C-(D/E+F)*G	A	+	B is operand, output B
*C-(D/E+F)*G	A B	+	* is operator, prec > +, push * into stack
C-(D/E+F)*G	A B	+ *	C is operand, output C
-(D/E+F)*G	A B C	+ *	- is operator, prec < *, pop *
-(D/E+F)*G	A B C *	+	- is operator, prec = +, pop +
-(D/E+F)*G	A B C * +	-	- is operator, prec > blank, push - into stack
(D/E+F)*G	A B C * +	- (	(, push
D/E+F)*G	A B C * + D	- (	D is operand, output D
/E+F)*G	A B C * + D	- (/	/ is operator, prec > (, push / into stack
E+F)*G	A B C * + D E	- (/	E is operand, output E
+F)*G	A B C * + D E /	- (	+ is operator, prec < /, pop /
+F)*G	A B C * + D E /	- (+	+ is operator, prec > (, push +
F)* G	A B C * + D E / F	- (+	F is operand, output F
)* G	A B C * + D E / F +	-	), pop +, pop and discard (, discard)
*G	A B C * + D E / F +	-	* is operator, prec > -, push *
G	A B C * + D E / F +	- *	G is operand, output G
empty	A B C * + D E / F + G	- *	No input remain, unstack all
empty	A B C * + D E / F + G * -	empty	

Activity 1

Exercise 4.1b:

Given an expression as

$$(A + B) * ((C - D) / (E + F)) * G$$

(a) Construct the expression tree.

(b) Convert into PRN postfix evaluation.

Activity 2

Exercise 4.2:

Get the infix expression for the following postfix:

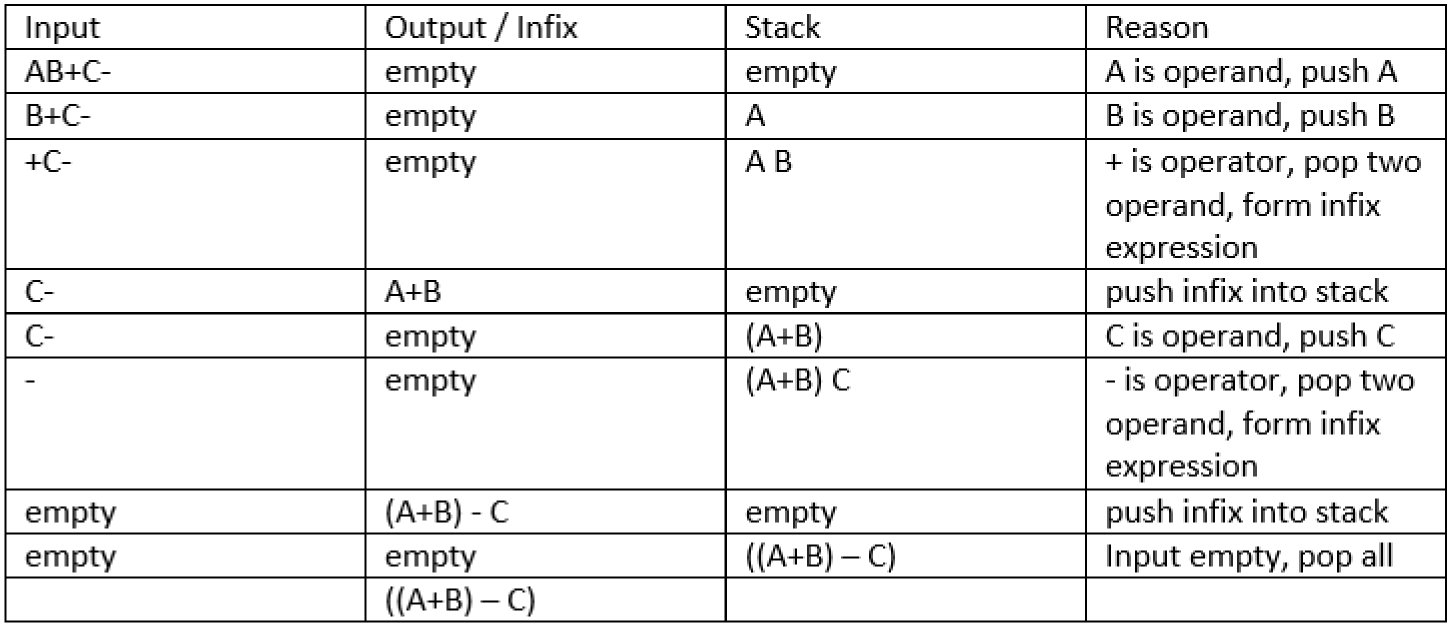
(a) $AB+C-$

(b) $AB+CD-*$

(c) $AB\wedge C*D-EF/GH+/+$

(d) $AB+C*DE--FG+\wedge$

(e) $ABCDE\wedge*/-$



Input	Output / Infix	Stack	Reason
AB+C-	empty	empty	A is operand, push A
B+C-	empty	A	B is operand, push B
+C-	empty	A B	+ is operator, pop two operand, form infix expression
C-	A+B	empty	push infix into stack
C-	empty	(A+B)	C is operand, push C
-	empty	(A+B) C	- is operator, pop two operand, form infix expression
empty	(A+B) - C	empty	push infix into stack
empty	empty	((A+B) – C)	Input empty, pop all
	((A+B) – C)		