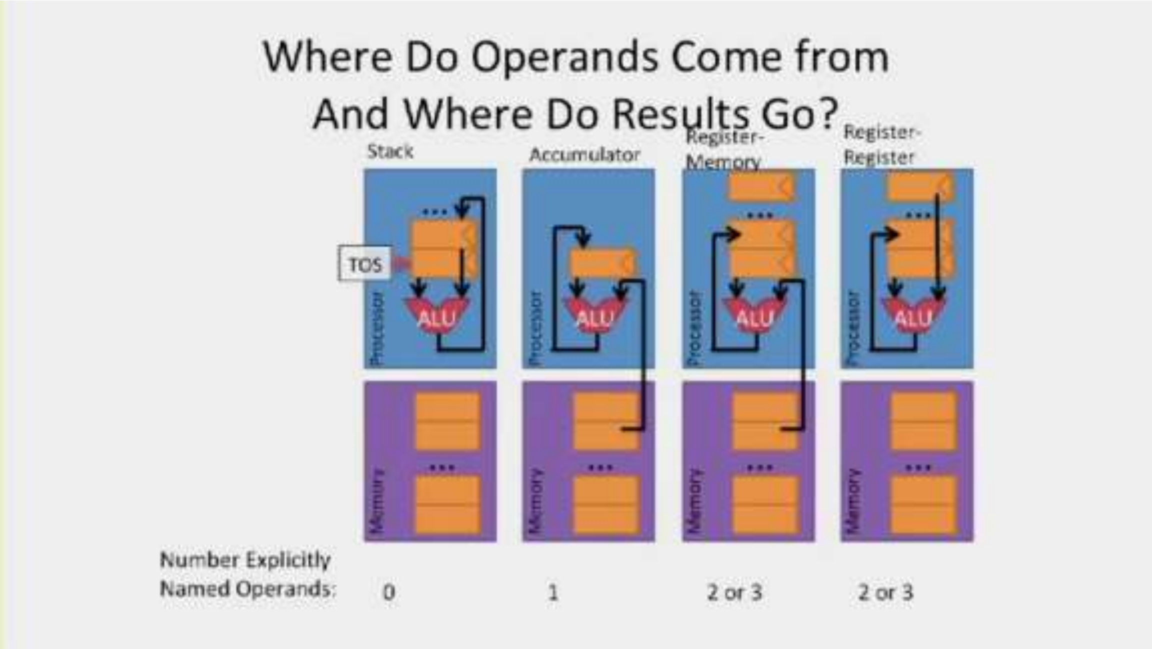
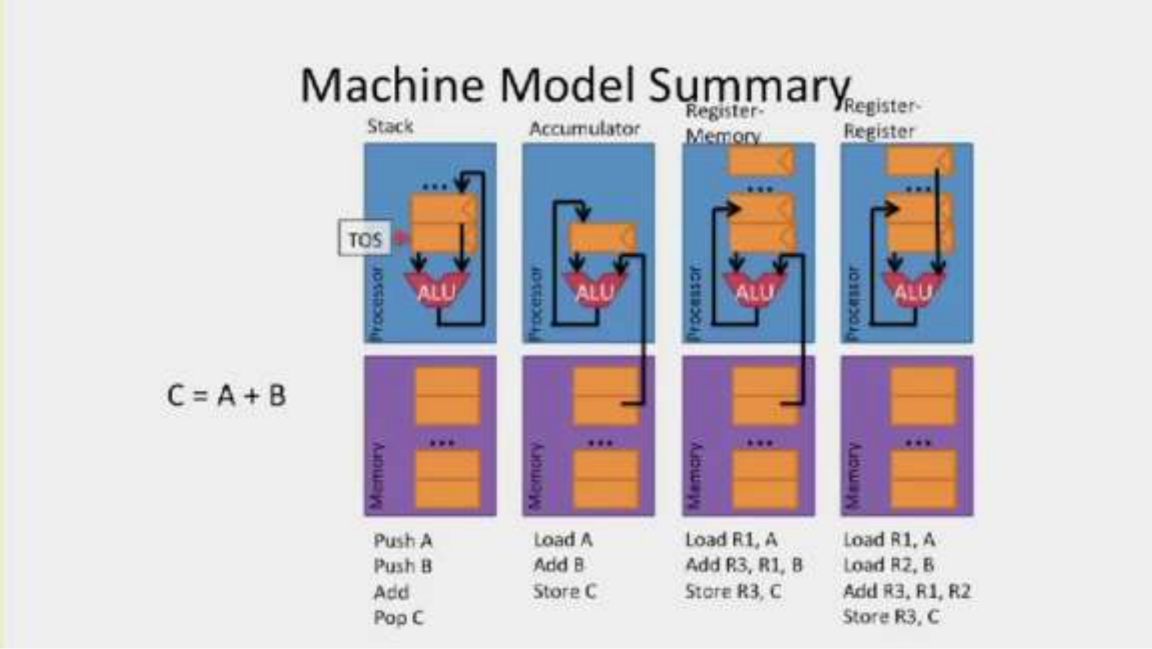
Prefetching and scalability



Where Do Operands Come from And Where Do Results Go?

	Stack	Accumulator	Register-Memory	Register-Register
Number Explicitly Named Operands:	0	1	2 or 3	2 or 3

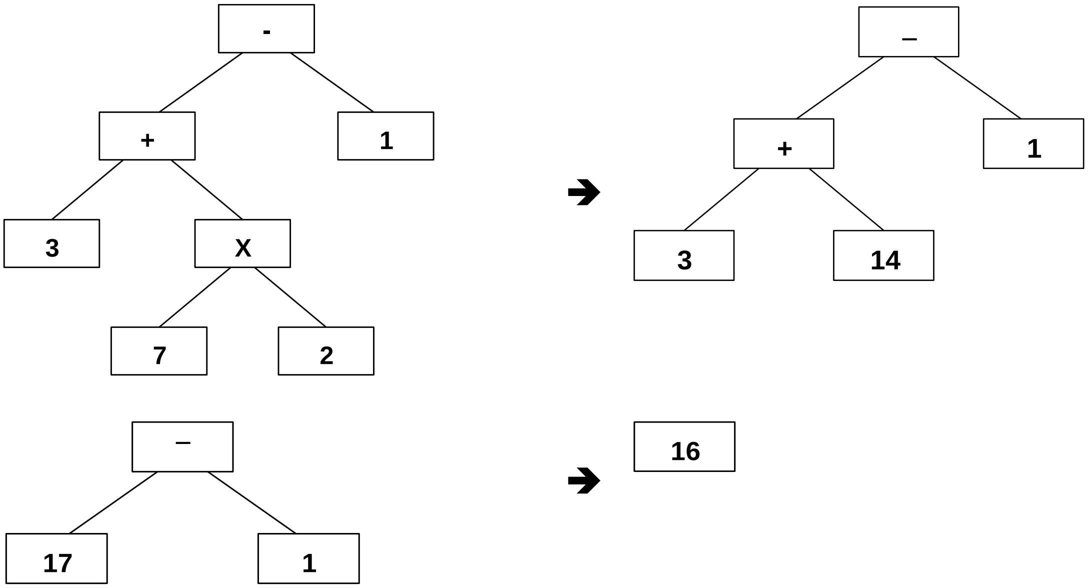


Machine Model Summary

C = A + B

Stack	Accumulator	Register-Memory	Register-Register
Push A	Load A	Load R1, A	Load R1, A
Push B	Add B	Add R3, R1, B	Load R2, B
Add	Store C	Store R3, C	Add R3, R1, R2
Pop C			Store R3, C

Expression tree for 3 + 7 × 2 – 1 = - + 3 × 7 2 1



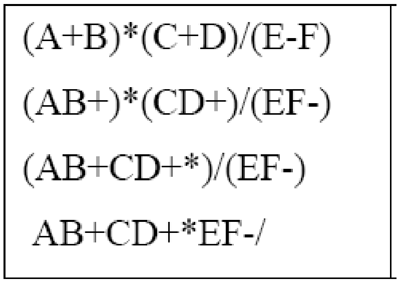
Create an expression tree for the following expression: 6 * 4 / 2 + 7 - 5 * 3

Reverse Polish Notation (RPN)

You will learn how to use a stack ADT to evaluate a mathematical expression written in reverse Polish notation (RPN= postfix notation).

(A+B)*C	AB+C*
infix	postfix

Notice that the **postfix** expression **HAS NO AMBIGUITY**, so it does not need **parentheses** to ensure that A+B is computed before the multiplication of C.



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

(AB+)*(CD+)/(EF-)

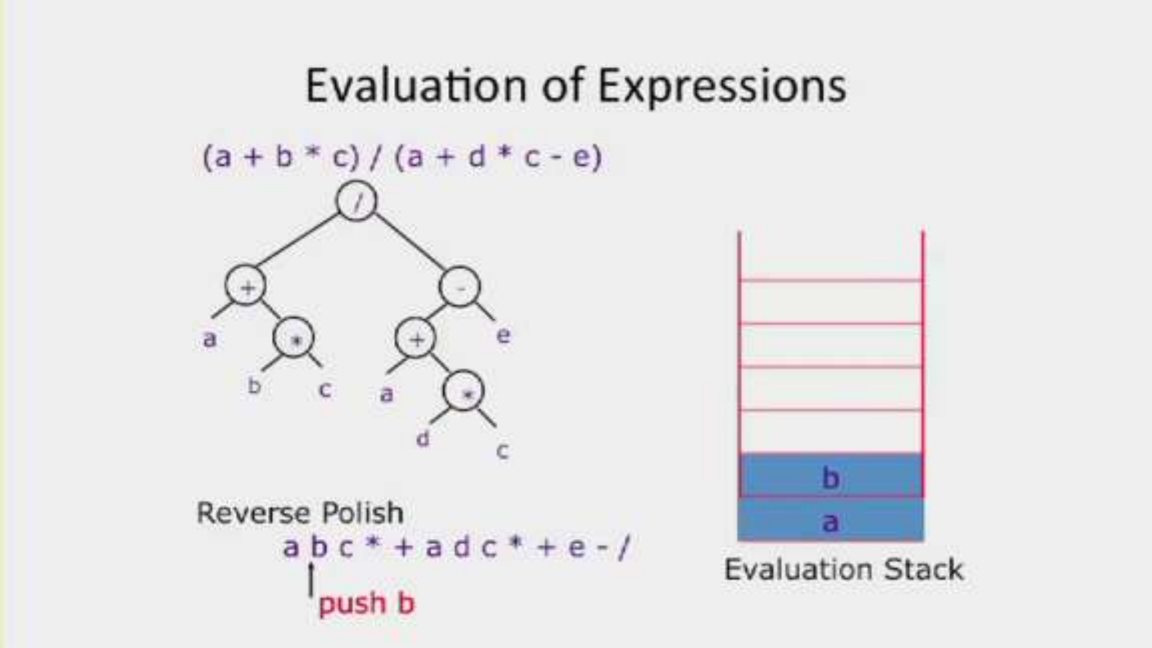
(AB+CD+*)/(EF-)

AB+CD+*EF-/

We can use a stack to evaluate a postfix expression in the following algorithm.
Scanning from left to right when we encounter an operand we push its value onto a stack. When we encounter a binary (two operand) operator we pop two values off the stack, perform the indicated operation and then push the result back onto the stack. When we are finished we can pop the stack to get the final result.

Enter values for three variables A=1, B=2 and C=3.

AB+C*	AB+C*	AB+C*	AB+C*	AB+C*
1	2 1	3	3 3	9
push A	push B	pop B pop A push A+B	push C	pop C pop A+B push (A+B)*C



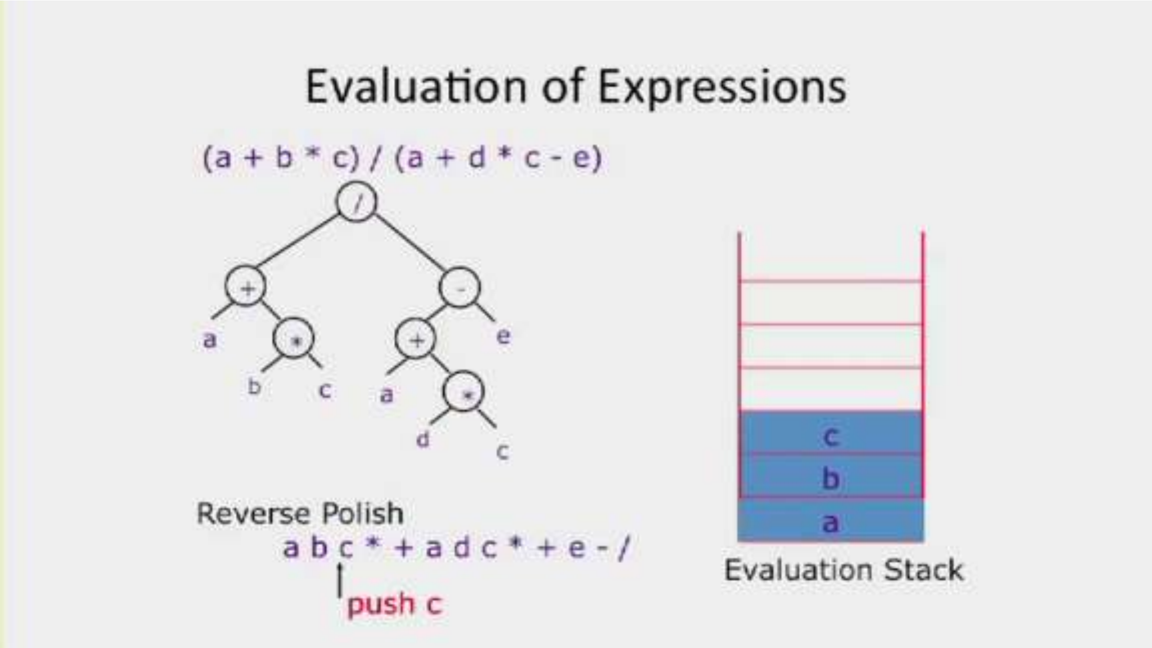
Evaluation of Expressions

(a + b * c) / (a + d * c - e)

Reverse Polish
a b c * + a d c * + e - /

push b

Evaluation Stack



Evaluation of Expressions

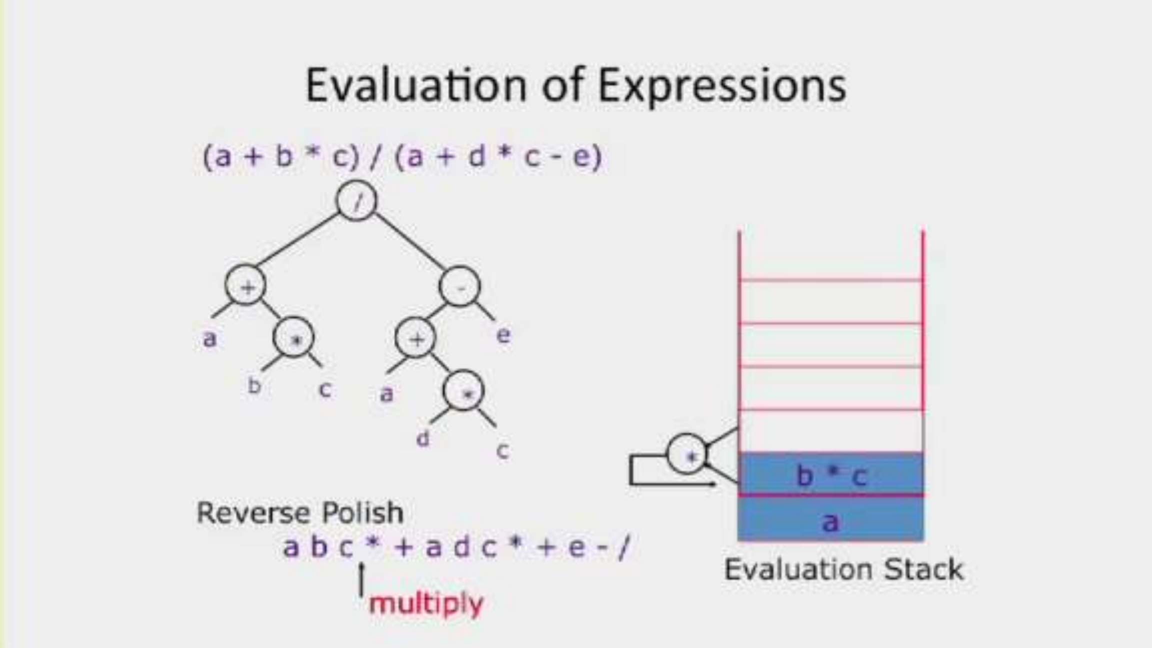
(a + b * c) / (a + d * c - e)

Evaluation Stack

Reverse Polish

a b c * + a d c * + e - /

push c



Evaluation of Expressions

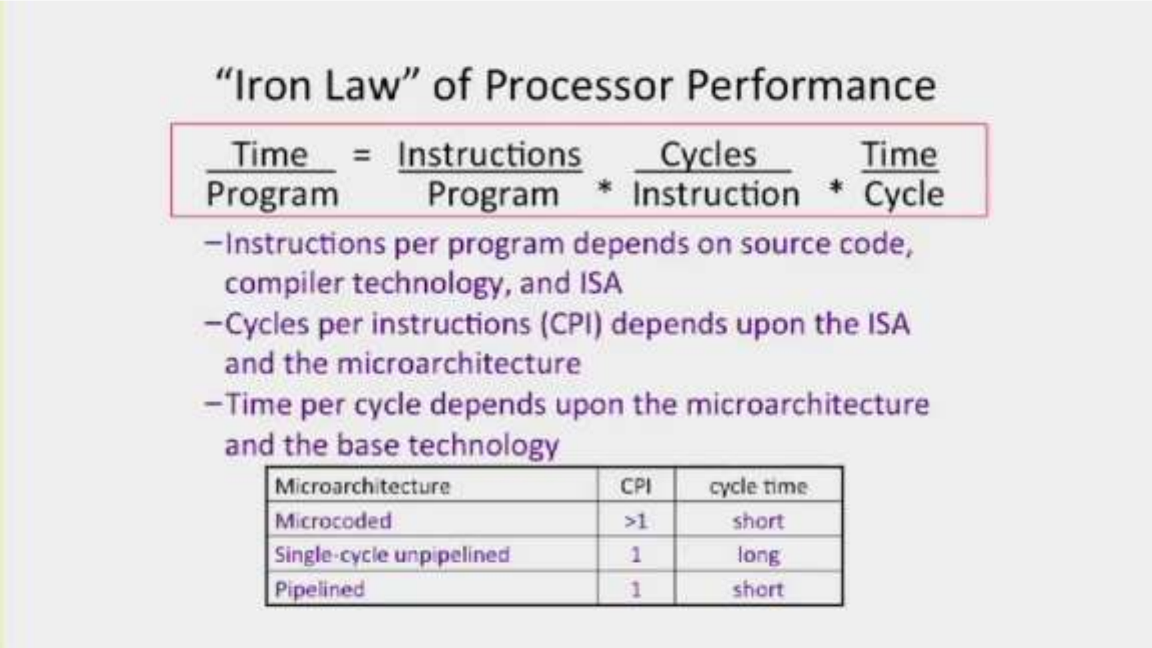
(a + b * c) / (a + d * c - e)

Evaluation Stack

Reverse Polish

a b c * + a d c * + e - /

multiply



“Iron Law” of Processor Performance

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Cycles}}{\text{Instruction}} * \frac{\text{Time}}{\text{Cycle}}$$

–Instructions per program depends on source code, compiler technology, and ISA

–Cycles per instructions (CPI) depends upon the ISA and the microarchitecture

–Time per cycle depends upon the microarchitecture and the base technology

Microarchitecture	CPI	cycle time
Microcoded	>1	short
Single-cycle unpipelined	1	long
Pipelined	1	short