Programming Techniques
Second year

Organization of Programming Languages

Understand how languages are designed and implemented

- **Syntax** -- What a program looks like
- **Semantics** -- What a program means
- **Implementation** -- How a program executes

Understand most appropriate language for solving specific problems, For example:

- **Pascal, C** -- procedural, statement oriented
- **C++, Java, Smalltalk** -- Object oriented
- **ML, Lisp** -- Functional
- **Prolog** -- Rule-based

Language Goals

- During 1950s--1960s - Compile programs to execute efficiently.
- There is a direct connection between language features and hardware - integers, reals, goto statements
- Programmers cheap; Machines expensive; Keep the machine busy

But today

- Compile programs that are built efficiently
- CPU power and memory very cheap
- Direct connection between language features and design concepts - encapsulation, records, inheritance, functionality, assertions

Why study programming languages?

- To improve your ability to develop effective algorithms
- To improve your use of your existing programming language
- To increase your vocabulary of useful programming constructs
- To allow a better choice of programming language
- To make it easier to learn a new language
- To make it easier to design a new language

Evolution of software architecture

- 1950s - Large expensive mainframe computers ran single programs (Batch processing)
- 1960s - Interactive programming (time-sharing) on mainframes
- 1970s - Development of Minicomputers and first microcomputers. Apple II. Early work on windows, icons, and PCs at XEROX PARC
- 1980s - Personal computer - Microprocessor, IBM PC and Apple Macintosh. Use of windows, icons and mouse
- 1990s - Client-server computing - Networking, The Internet, the World Wide Web

Attributes of a good language

- Clarity, simplicity, and unity - provides both a framework for thinking about algorithms and a means of expressing those algorithms
- Orthogonality -every combination of features is meaningful
- Naturalness for the application - program structure reflects the logical structure of algorithm
- Support for abstraction - program data reflects problem being solved

Attributes of a good language (continued)

- Ease of program verification - verifying that program correctly performs its required function
- Programming environment - external support for the language
- Portability of programs - transportability} of the resulting programs from the computer on which they are developed to other computer systems
- Cost of use - program execution, program translation, program creation, and program maintenance

Language paradigms

Imperative languages

- Goal is to understand a machine state (set of memory locations, each containing a value)
- Statement oriented languages that change machine state (C, Pascal, FORTRAN, COBOL)
- Syntax: S1, S2, S3, ...

Applicative (functional) languages

- Goal is to understand the function that produces the answer
- Function composition is major operation (ML, LISP)
- Syntax: P1(P2(P3(X)))
- Programming consists of building the function that computes the answer

Language paradigms (continued)

Rule-based languages

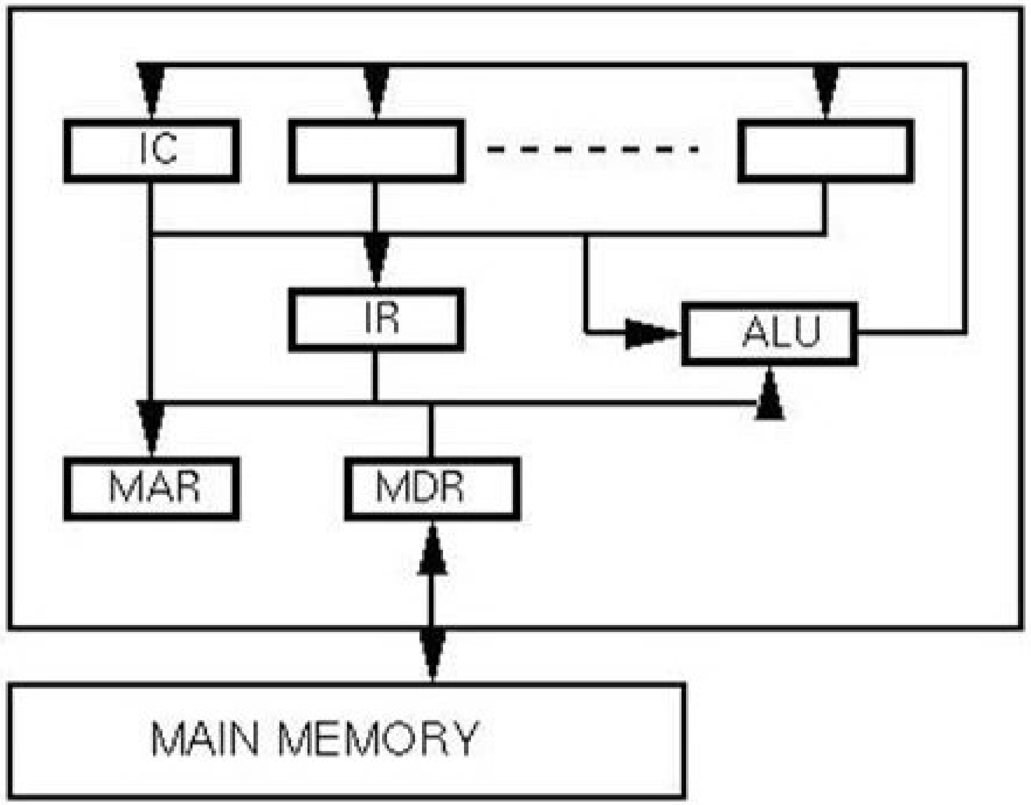
- Specify rule that specifies problem solution (Prolog, BNF Parsing)
- Other examples: Decision procedures, Grammar rules (BNF)
- Syntax: Answer $\rightarrow$ specification rule
- Programming consists of specifying the attributes of the answer

Object-oriented languages

- Imperative languages that merge applicative design with imperative statements (Java, C++, Smalltalk)
- Syntax: Set of objects (classes) containing data (imperative concepts) and methods (applicative concepts)

Machine architecture

Typical machine design



← Many high speed registers

← Arithmetic / Logic unit

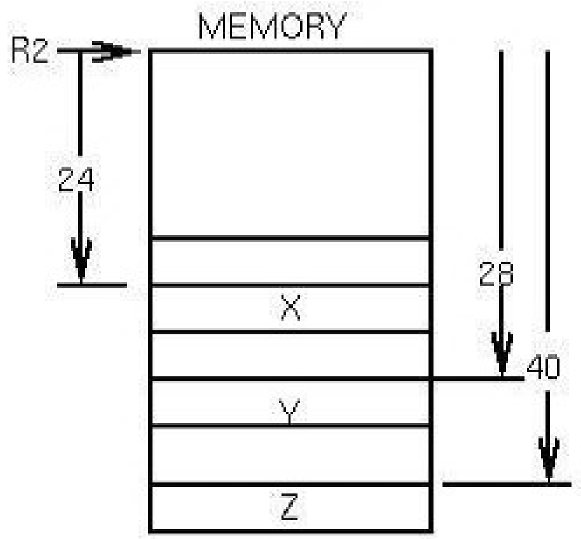
IC – Instruction counter
MAD – Memory address register
MDR– Memory data register
IR – Instruction register

Two cycles:

•Fetch cycle - get instruction

•Execute cycle - do operation

Typical machine translation



Instruction format:

```
        Opcode  register, index, offset
        load    R1, R2, 24
```

For example in C: As we see later, memory for data in blocks of storage pointed to by a register:

```
    X = Y + Z
```

could be translated as:

```
  load      R1, R2, 28    [Location of Y]
  add       R1, R2, 40    [Location of Z]
  store     R1, R2, 24    [Location of X]
```

Software architectures

Previously

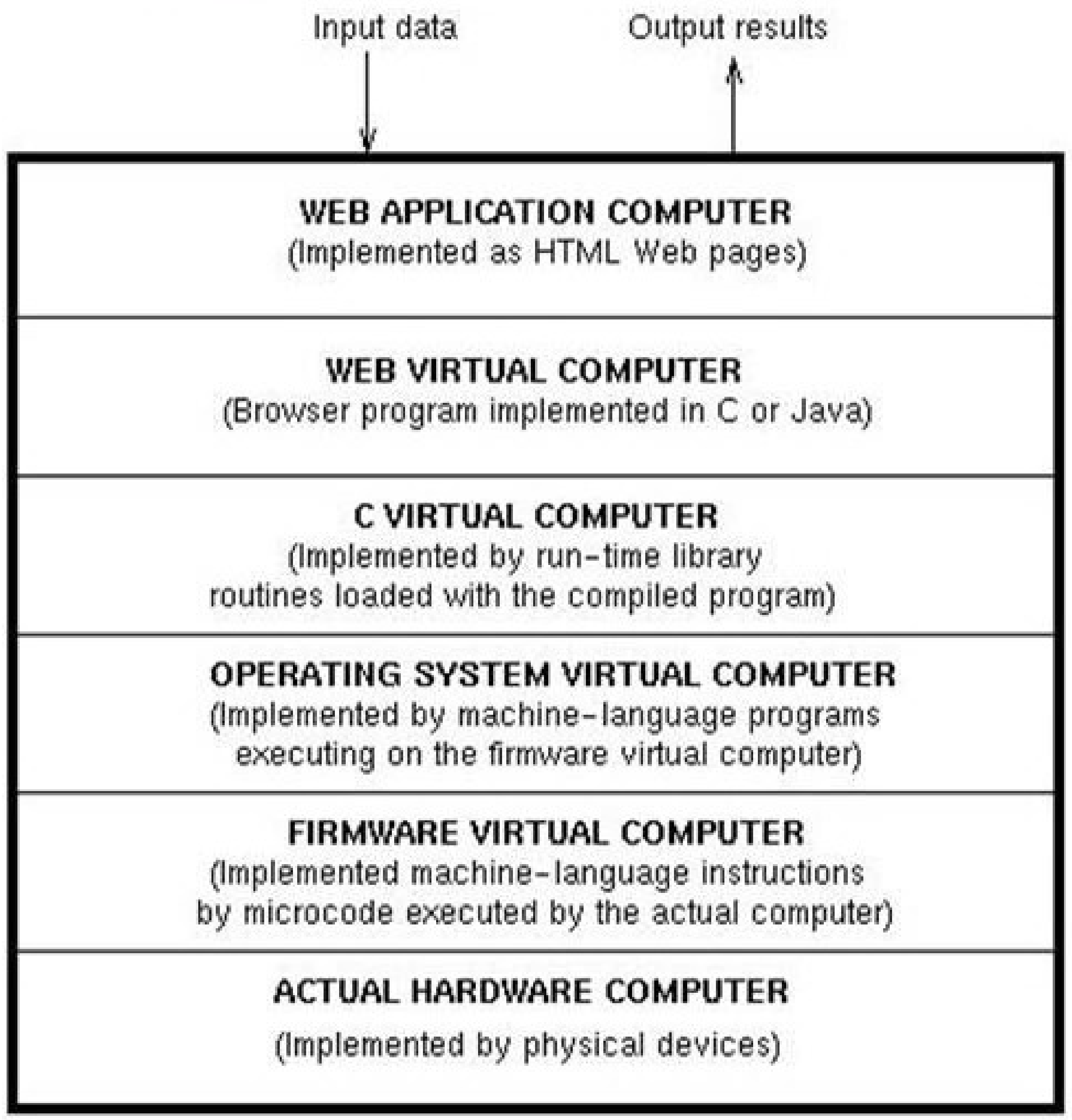
- Build program to use hardware efficiently.
- Often use of machine language for efficiency.

Today

- No longer write directly in machine language.
- Use of layers of software.
- Concept of *virtual machines*. Each layer is a machine that provides functions for the next layer.

Virtual Machines

Example: Web application


Input data
Output results
WEB APPLICATION COMPUTER
(Implemented as HTML Web pages)
WEB VIRTUAL COMPUTER
(Browser program implemented in C or Java)
C VIRTUAL COMPUTER
(Implemented by run-time library routines loaded with the compiled program)
OPERATING SYSTEM VIRTUAL COMPUTER
(Implemented by machine-language programs executing on the firmware virtual computer)
FIRMWARE VIRTUAL COMPUTER
(Implemented machine-language instructions by microcode executed by the actual computer)
ACTUAL HARDWARE COMPUTER
(Implemented by physical devices)

Language translation

Program structure

Syntax

- What a program looks like
- BNF (context free grammars) - a useful notation for describing syntax.

Semantics

- Execution behavior
- Static semantics - Semantics determined at compile time:
  - var A: integer; Type and storage for A
  - int B[10]; Type and storage for array B
  - float MyProcC(float x;float y){...}; Function attributes
- Dynamic semantics - Semantics determined during execution:
  - X = ``ABC'' X a string
  - X = 1 + 2; X an integer

Aspects of a program

Declarations - Information for compiler

- `var A: integer;`
- `typedef struct { int A; float B } C;`

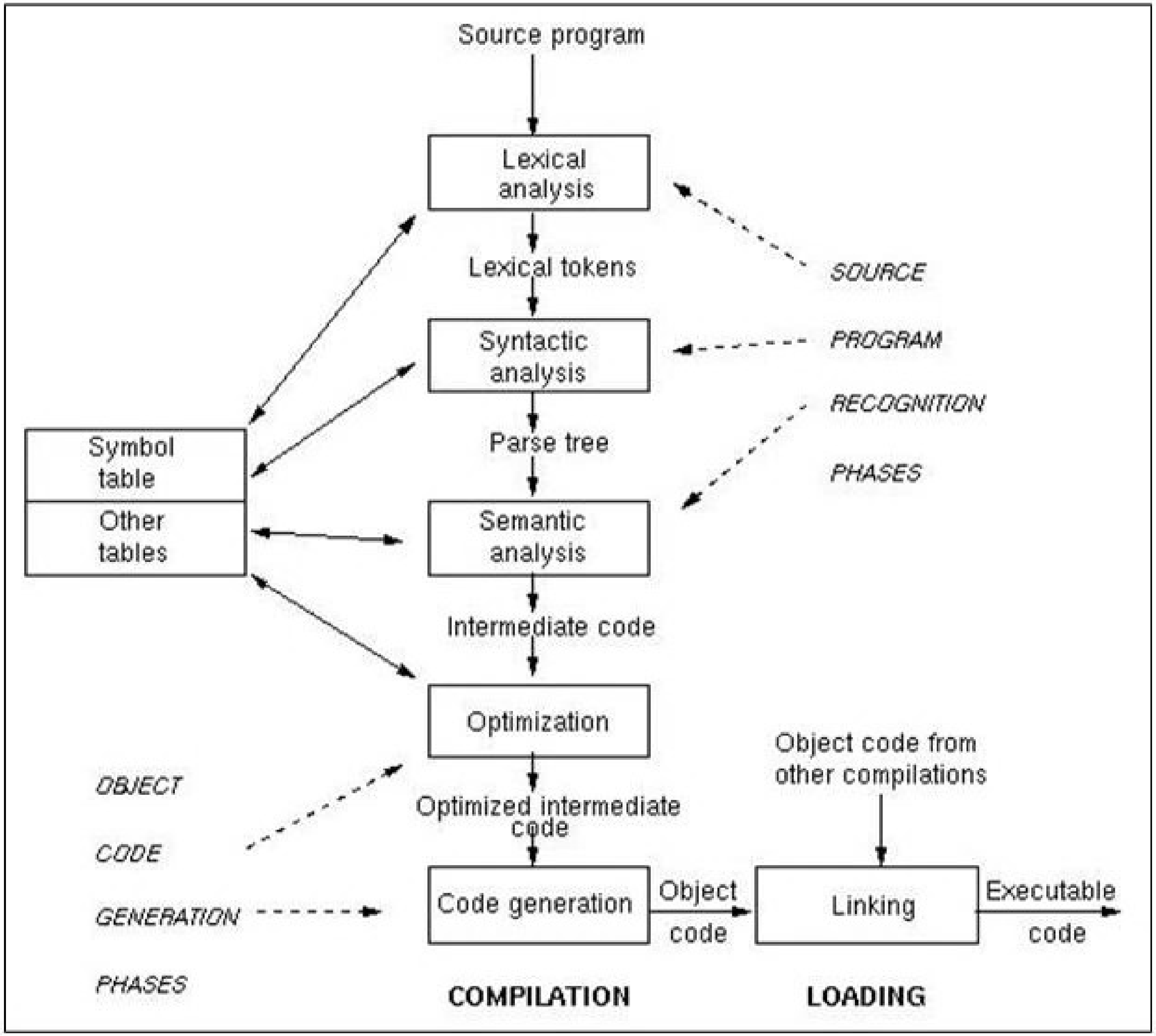
Control - Changes to state of the machine

- `if (A<B) { ... }`
- `while (C>D) { ... }`

Structure often defined by a Backus Naur Form (*BNF*) grammar (First used in description of Algol in 1958. Peter Naur was chair of Algol committee, and John Backus was secretary of committee, who wrote report.)

We will see later - BNF turns out to be same as context free grammars developed by Noam Chomsky, a linguist)

Stages in translating a program



Major stages

Lexical analysis (Scanner): Breaking a program into primitive components, called tokens (identifiers, numbers, keywords, ...)

Syntactic analysis (Parsing): Creating a syntax tree of the program.

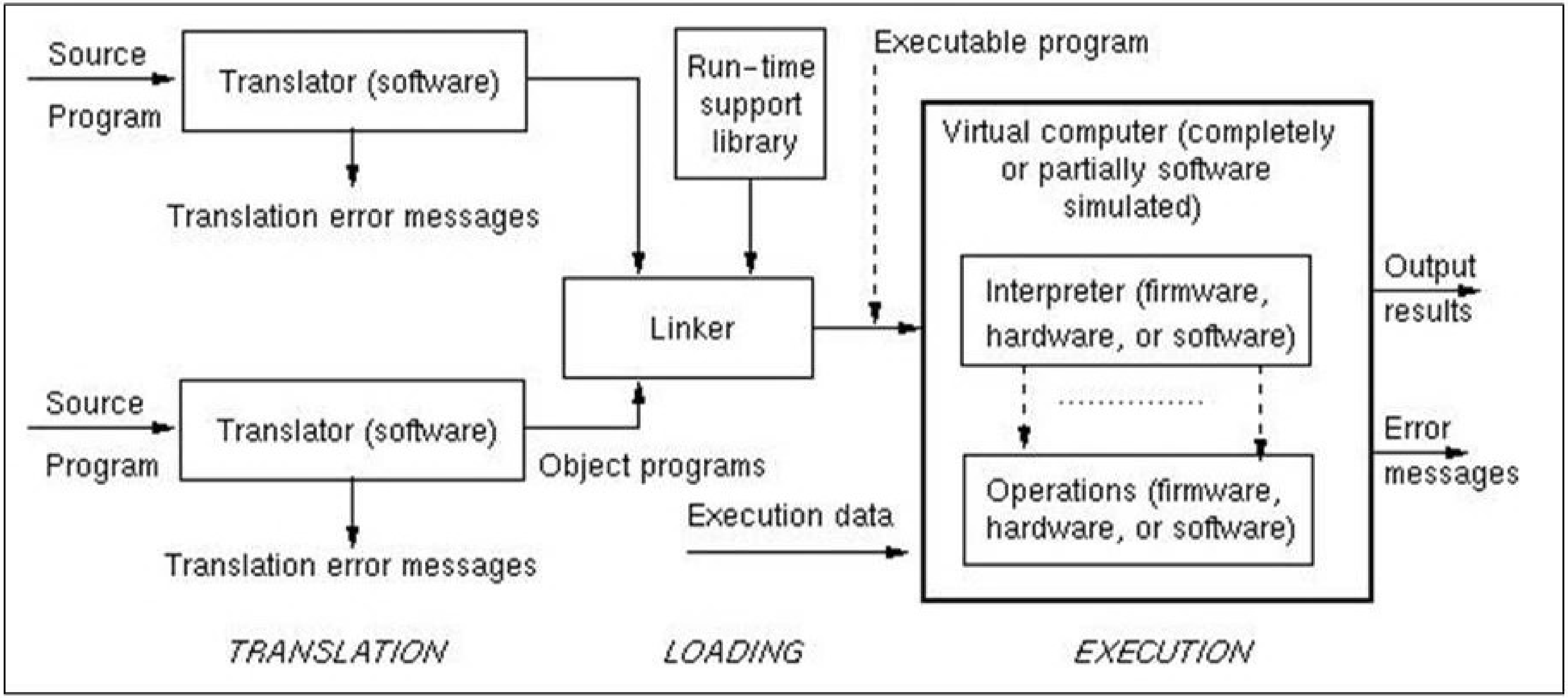
Symbol table: Storing information about declared objects (identifiers, procedure names, ...)

Semantic analysis: Understanding the relationship among the tokens in the program.

Optimization: Rewriting the syntax tree to create a more efficient program.

Code generation: Converting the parsed program into an executable form.

Translation environments


Source Program
Translator (software)
Translation error messages
Source Program
Translator (software)
Translation error messages
Object programs
Run-time support library
Linker
Executable program
Execution data
Virtual computer (completely or partially software simulated)
Interpreter (firmware, hardware, or software)
Operations (firmware, hardware, or software)
Output results
Error messages
TRANSLATION
LOADING
EXECUTION

Expressions

Arithmetic Operators
Rules of Operator Precedence

Operator(s)	Precedence & Associativity
( )	Evaluated first. If nested (embedded), innermost first. If on same level, left to right.
* / %	Evaluated second. If there are several, evaluated left to right.
+ -	Evaluated third. If there are several, evaluated left to right.
=	Evaluated last, right to left.

Using Parentheses

- Use parentheses to change the order in which an expression is evaluated.

```
a + b * c Would multiply b * c first,
          then add a to the result.
```

- If you really want the sum of a and b to be multiplied by c, use parentheses to force the evaluation to be done in the order you want.

```
(a + b) * c
```

- Also use parentheses to clarify a complex expression.

Extended Example

- Given integer variables a, b, c, d, and e, where a = 1, b = 2, c = 3, d = 4, evaluate the following expression:

```
e = b % d / c * b - a
e = ( b % d ) / c * b - a
e = ( ( b % d ) / c ) * b - a
e = ( ( ( b % d ) / c ) * b ) - a
e = ( ( ( ( b %d ) / c ) * b ) - a )
e = ( ( ( ( 2 % 4 ) / 3 ) * 2 ) - 1)
```

Extended Example (cont'd)

```
e = ( ( ( ( 2 % 4 ) / 3 ) * 2 ) - 1 )
e = ( ( ( ( 2 ) / 3 ) * 2 ) - 1 )
e = ( ( ( 2 / 3 ) * 2 ) - 1 )
e = ( ( ( 0 ) * 2 ) - 1 )
e = ( (  0  * 2 ) - 1 )
e = ( (  0 ) - 1 )
e = ( 0  - 1 )
e =  - 1
```

```
Note:  Always use parenthesis when you
  have more than two operators!
```

Another Extended Example

Given integer variables a, b, c, d, and e, where a = 1, b = 2, c = 3, d = 4, evaluate the following expression:

```
d = e = 1 + a + b * d % c
d = e = 1 + a + ( b * d ) % c
d = e = 1 + a + ( ( b * d ) % c )
d = e =( 1 + a ) + ( ( b * d ) % c )
d = e =(  ( 1 + a ) + (( b * d ) % c ))
d =(e =(( 1 + a ) + (( b * d ) % c )) )
d =(e =(  ( 1 + 1 ) + (( 2 * 4 ) % 3) ))
```

Another Extended Example (cont'd)

```
d =(e =  (  ( 1 + 1 ) + (( 2 * 4 ) % 3 )) )
d = ( e = ( ( 1 + 1 ) + ( ( 8 ) % 3 ) ) )
d = ( e = ( ( 1 + 1 ) + ( 8 % 3 ) ) )
d = ( e = ( ( 1 + 1 ) + ( 2 ) ) )
d = ( e = ( ( 1 + 1 ) + 2 ) )
d = ( e = ( ( 2 ) + 2 ) )
d = ( e = ( 2 + 2 ) )
d = ( e = ( 4 ) )
d = ( e = 4 )

d = 4  /*e is now set to 4 and so is d */
```

Practice With Evaluating Expressions

Given integer variables a, b, c, d, and e, where a = 1, b = 2, c = 3, d = 4, evaluate the following expressions:

```
a + b - c + d
a * b / c
1 + a * b % c
a + d % b - c
e = b = d + c / b - a
```

Postfix

Infix notation: Operator appears between operands:

2 + 3 → 5

3 + 6 → 9

Implied precedence: 2 + 3 * 4 → 2 + (3 * 4),
not (2 + 3) * 4

Prefix notation: Operator precedes operands:

+ 2 3 → 5

+ 2 * 3 5 → (+ 2 (* 3 5)) → + 2 15 → 17

Postfix notation: Operator follows operands:

2 3 + → 5

2 3 * 5 + →((2 3 * 5 +) → 6 5 + → 11

Called Polish postfix since few could pronounce Polish mathematician Lukasiewicz, who invented it.

An interesting, but unimportant mathematical curiosity when presented in 1920s. Only became important in 1950s when Burroughs rediscovered it for their ALGOL compiler.

Evaluation of postfix

1. If argument is an operand, stack it.
2. If argument is an n-ary operator, then the n arguments are already onthe stack. Pop the n arguments from the stack and replace by the value of the operator applied to the arguments.

Example: 2 3 4 + 5 * +

1. 2 - stack
2. 3 - stack
3. 4 - stack
4. + - replace 3 and 4 on stack by 7
5. 5 - stack
6. * - replace 5 and 7 on stack by 35
7. + - replace 35 and 2 on stack by 37

Importance of Postfix to Compilers

Code generation same as expression evaluation.

To generate code for 2 3 4 + 5 * +, do:

1. 2 - stack L-value of 2
2. 3 - stack L-value of 3
3. 4 - stack L-value of 4
4. + - generate code to take R-value of top stack element (L-value of 4) and add to R-value of next stack element (L-value of 3) and place L-value of result on stack
5. 5 - stack L-value of 5
6. * - generate code to take R-value of top stack element (L-value of 5) and multiply to R-value of next stack element (L-value of 7) and place L-value of result on stack
7. + - generate code to take R-value of top stack element (L-value of 35) and add to R-value of next stack element (L-value of 2) and place L-value of result (37) on stack

Parameter transmission

Parameter passing

Parameter: A variable in a procedure that represents some other data from the procedure that invoked the given procedure.

Parameter transmission: How that information is passed to the procedure.

- The parameter is also called the formal argument. The data from the invoking procedure is called the actual argument or sometimes just the argument.

Usual syntax:

Actual arguments: call P(A, B+2, 27+3)

Parameters: Procedure P(X, Y, Z)

What is connection between the parameters and the arguments?

- Call by name
- Call by reference
- Call by value
- Call by result (or value-result)

Language dependent

Difference languages have different mechanisms:

- ALGOL - name, value
- Pascal - value, reference
- C - value (BUT pointers give us reference

Constant tension between desire for efficiency and semantic correctness in defining parameter transmission.

Call by name

Substitute argument for parameter at each occurrence of parameter:

```
Invocation: P(A, B+2, 27+3)
Definition: procedure P(X,Y,Z)
                      {int I; I=7; X = I + (7/Y)*Z;}
Meaning: P(X,Y,Z) {int I; I=7; A=I+(7/(B+2))*(27+3);}
```

This is a true macro expansion. Simple semantics, BUT:

1. Implementation. How to do it?
2. Aliases. What if statement of P were: I = A?
3. Expressions versus statements: If we had D=P(1,2,3) and a return(42) in P, what does semantics mean?
4. Error conditions: P(A+B, B+2, 27+3)

Implementation of call by name

A thunk is the code which computes the L-value and R-value of an argument.

For each argument, pass code address that computes both L-values and R-values of arguments.

P(A, B+2, 27+3) generates:

```
jump to subroutine P
address of thunk to return L-value(A)
address of thunk to return R-value(A)
address of thunk to return L-value(B+2)
address of thunk to return R-value(B+2)
address of thunk to return L-value(27+3)
address of thunk to return R-value(27+3)
```

To assign to X, call thunk 1, To access X, call thunk 2

To assign to Y, call thunk 3, To access Y, call thunk 4

To assign to Z, call thunk 5, To access Z, call thunk 6

Issue: Assignment to (B+2): How?

Call by name is conceptually convenient, but inefficient.

Examples of Call by Name

```
1. P(x) {x = x + x;}
   Seems simple enough …
        Y = 2; P(Y); write(Y) ⇒ means Y = Y+Y
        write(Y) ⇒ prints 4
2. int A[10];
   for(I=0; I<10; I++) {A[I]=I;};
   I=1; P(A[I]) ⇒ A[1] = A[1] + A[1] ⇒ A[1] set to 2
3. But: F {I = I + 1; return I;}
   What is: P(A[F])?
   P(A[F]) ⇒ A[F] = A[F]+A[F] ⇒ A[I++] = A[I++]+A[I++]
           ⇒ A[2] = A[3]+A[4]
4. Write a program to exchange values of X and Y:
   (swap(X,Y))
Usual way: swap(x,y) {t=x; x=y; y=t;}
Cannot do it with call by name. Cannot handle both of
   following: swap(I, A[I]) swap(A[I],I)
One of these must fail.
```

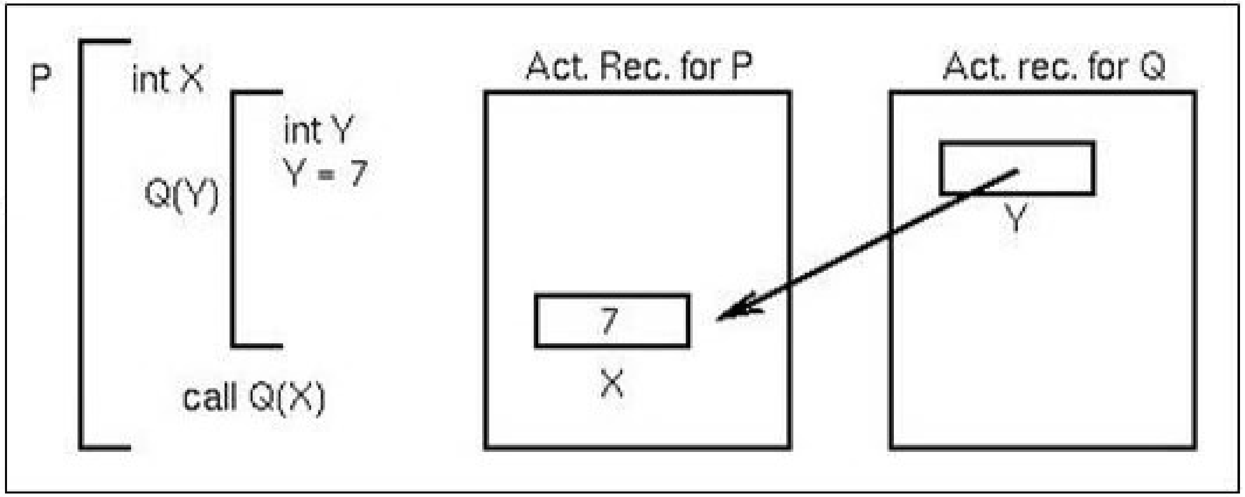
Call by reference

Pass the L-value of the argument for the parameter.

Invocation: P(A, B+2, 27+3)

Implementation:

```
Temp1 = B+2
Temp2 = 27+3
jump to subroutine P
L-value of A
L-value of Temp1
L-value of Temp2
```



This is the most common parameter transmission mechanism. In the procedure activation record, parameter X is a local variable whose R-value is the L-value of the argument.

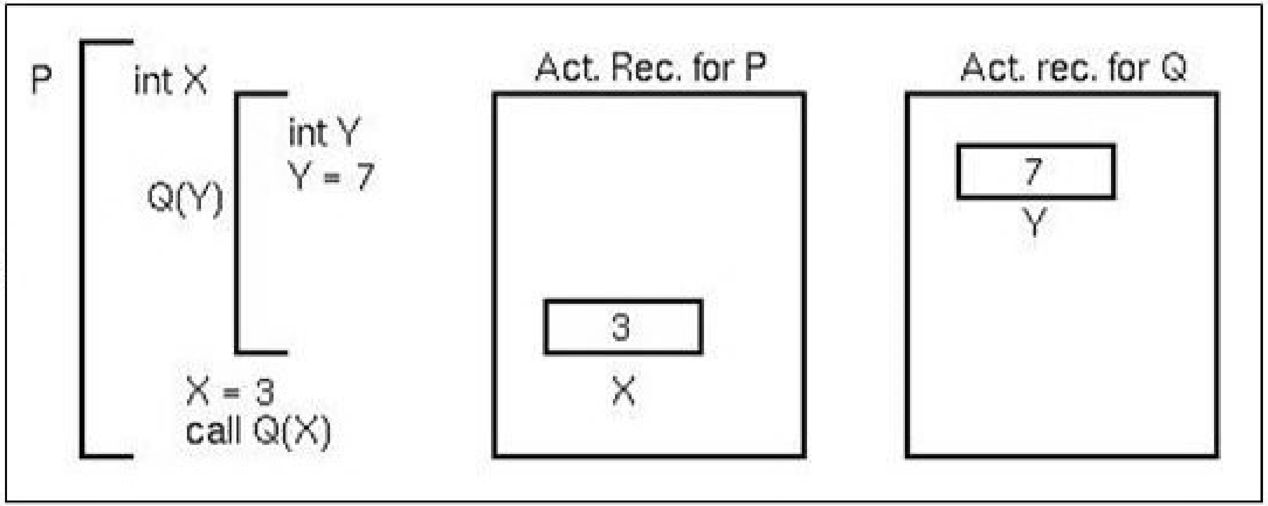
Call by value

Pass the R-value of the argument for the parameter.

Invocation: P(A, B+2, 27+3)

Implementation:

```
   Temp1 = B+2
   Temp2 = 27+3
   jump to subroutine P
   R-value of A
   R-value of Temp1
   R-value of Temp2
```



In procedure activation record, parameter X is a local variable whose R-value is the R-value of the argument.

Call by reference in C

C only has call by value,

BUT pointer variables allow for simulating call by reference:

P(i, j) ⇒ passes i and j by value.

P(&i, &j) ⇒ passes L-values of i and j.

P(*x, *y) {*x = *y + 1;} ⇒ arguments are addresses (pointers)

Call by result (or value-result): Call by value, AND pass back the final value to argument upon return.

Parameter is a local value in procedure.

Similar to call by reference, except for aliasing.

In-out semantics

Parameters in Ada are based upon use (semantics), not implementation:

in - argument value will be used in procedure.

out - parameter value will be used in calling program.

in out - both uses of arguments and parameters

```
P(X in integer;
    Y out integer;
    Z in out integer);
begin ... end;
```

In Ada 83, language definition allowed some latitude in implementation $\Rightarrow$ as long as implementation consistent, ok.

But this meant that the same program could give different answers from different standards conforming compilers

In Ada 95, more restricted: in integer is value, out integer is value-result, composite (e.g., arrays) is reference.

Example of parameter passing

```
Main
{A = 2; B = 5; C = 8; D = 9;
   P(A, B, C, D); write(A, B, C, D);
P(U, V, W, X)
   {V = U+A;
    W = A+B;
    A = A+1;
    X = A+2;
    write(U, V, W, X)}
Fill in table assume parameters are of the given type:
```

	A	B	C	D	U	V	W	X	print P	print main
Call by name										
Call by reference										
Call by value										
Call by result										

```
When do call by name and call by reference differ?
    When L-value can change between parameter
    references. E.g., P(I, A[I])
```

Parallel execution

Parallel programming principles

Variable definitions. Variables may be either mutable or definitional. Mutable variables are the common variables declared in most sequential languages. Values may be assigned to the variables and changed during program execution. A definitional variable may be assigned a value only once.

Parallel composition. We need to add the parallel statement, which causes additional threads of control to begin executing.

Program structure. They may be transformational to transform the input data into an appropriate output value. Or it may be reactive, where the program reacts to external stimuli called events.

Communication. Parallel programs must communicate with one another. Such communication will typically be via shared memory with common data objects accessed by each parallel program or via messages.

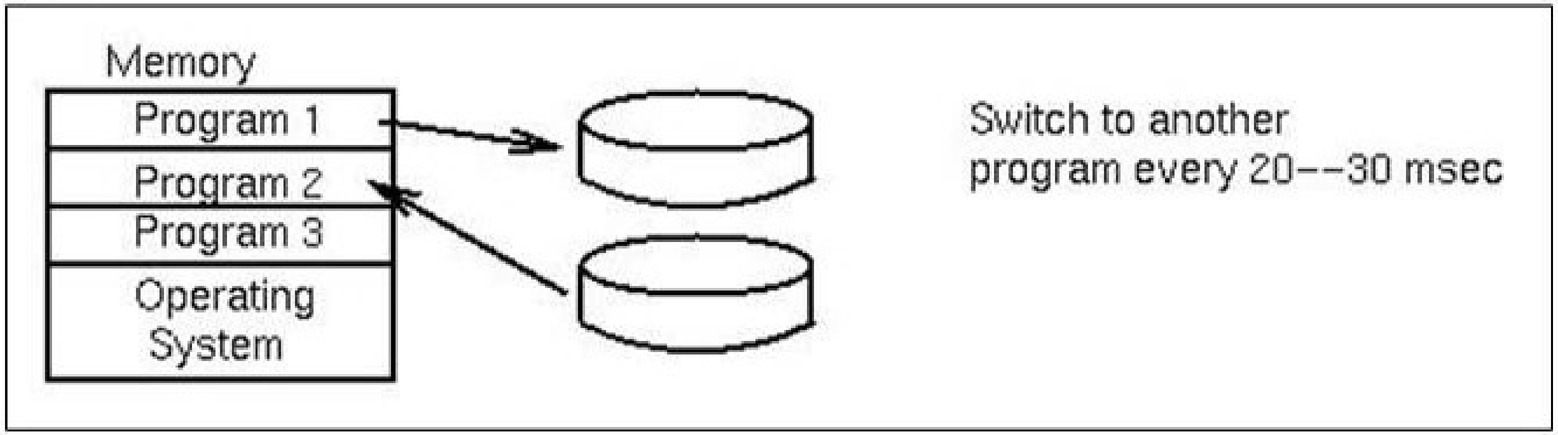
Synchronization. Parallel programs must be able to order the execution of its various threads of control.

Impact of slow memories

Historically - CPU fast

Disk, printer, tape - slow

What to do while waiting for I/O device? - Run another program:



Even today, although machines and memory are much faster, there is still a 10^5 or more to 1 time difference between the speed of the CPU and the speed for accessing information from disk. For example,

- Instruction time: 50 nanosecond
- Disk access: 10 milliseconds = 10,000,000 nanoseconds

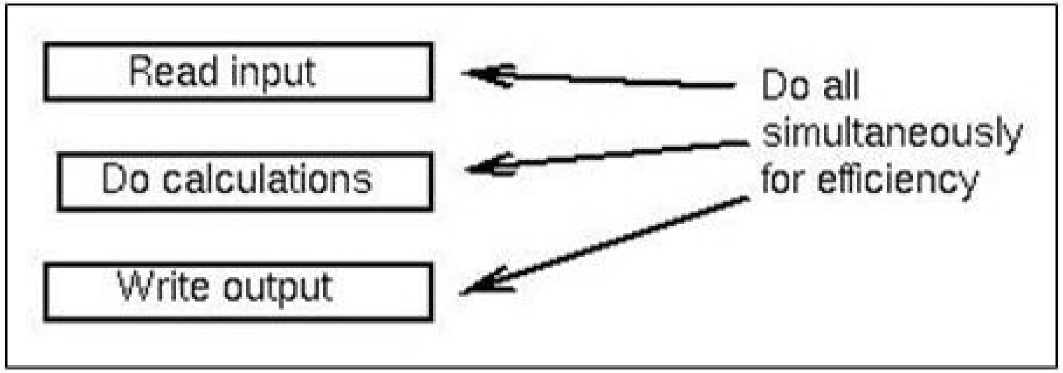
Multiprogramming

Now:

Multiple processors

Networks of machines

Multiple tasks simultaneously



Problems:

1. How to switch among parts effectively?

2. How to pass information between 2 segments?

Content switching of environments permitting concurrent execution of separate programs.

Parallel constructs

Two approaches (of many):

1. AND statement (programming language level)
2. fork function (UNIX) (operating system level)

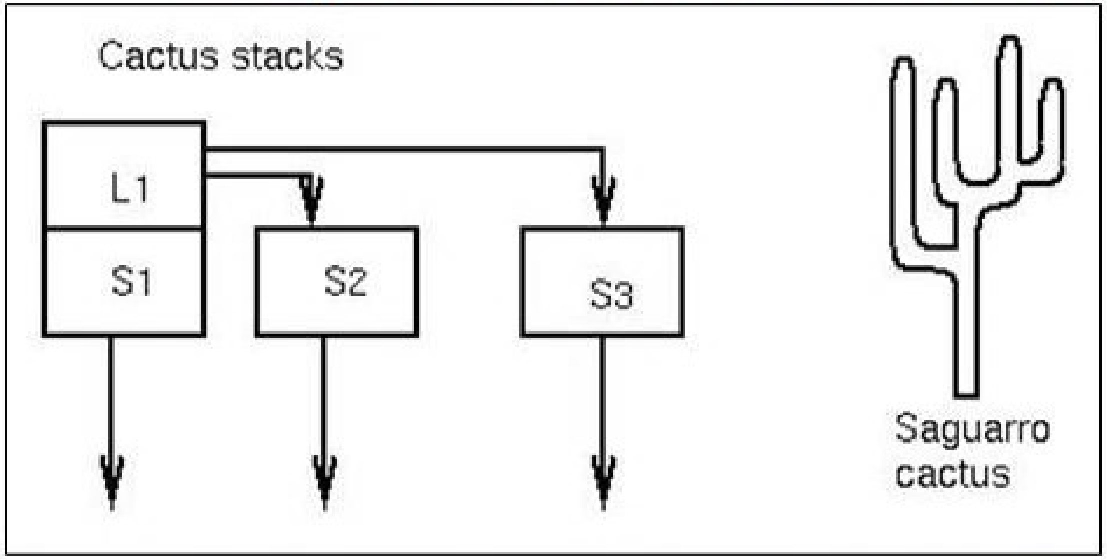
and: Syntax: statement1 and statement2 and statement3

Semantics: All statements execute in parallel.

Execution goes to statement following *and* after all parallel parts terminate.

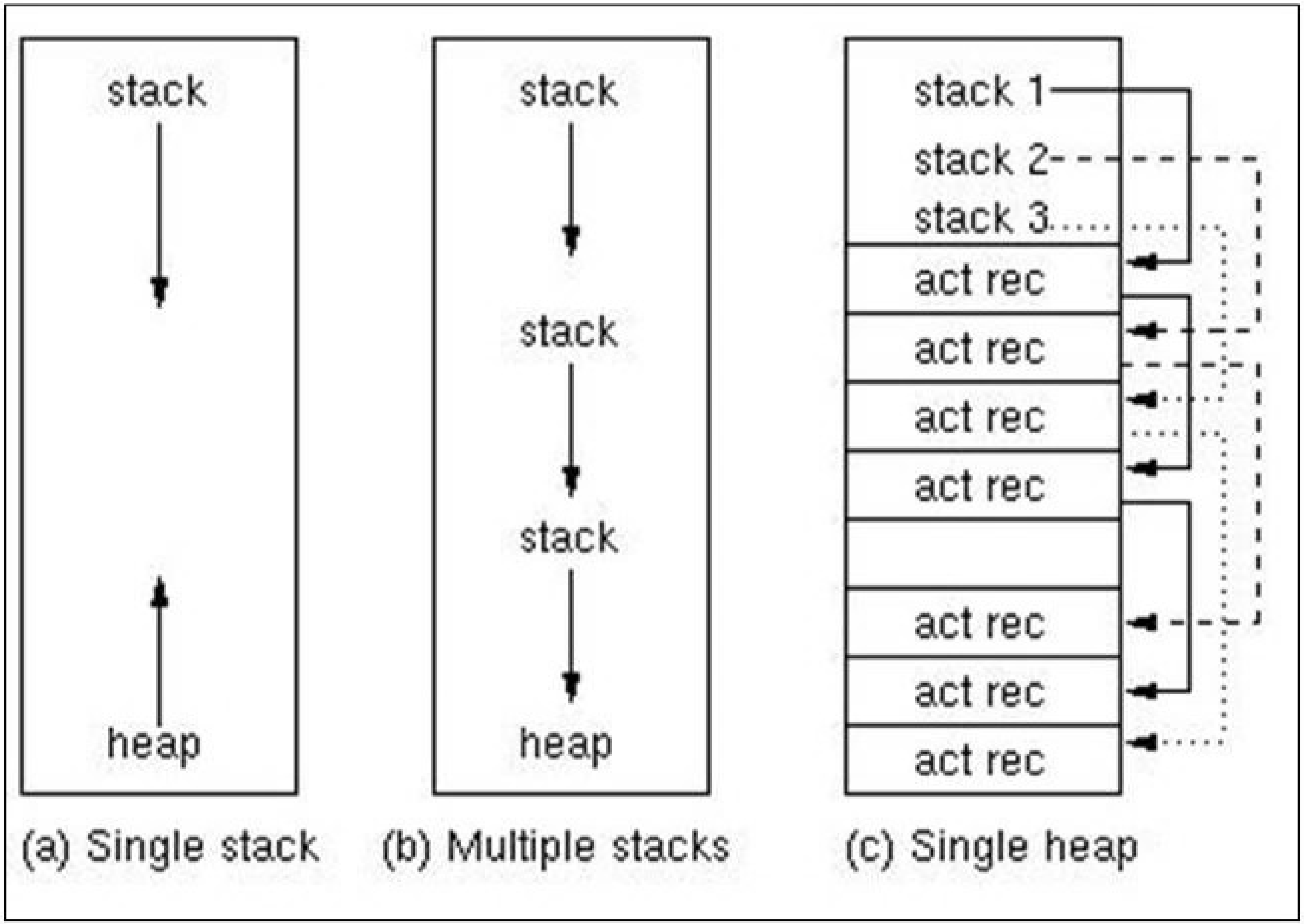
S1; S1 and S2 and S3; S4 ⇒ S4 after S1, S2, and S3 terminate

⇒ Implementation: Cactus stack



Parallel storage management

```
Use multiple stacks. Can use one heap (c)
```



“and” statement execution

After L1, add S1, S2, S3 all onto stack.

Each stack is independent.

How to implement? ⇒ Heap storage is one way for each activation record.

2. fork() function:

```
{ S1; fork();
    if I am parent process
        do { main task;
            sleep until child process terminates
    if I am child process do { exec new process
S2  ⇒ S2 executes when both parent and child
          process terminate above action
```

Both parent process and child process execute independently

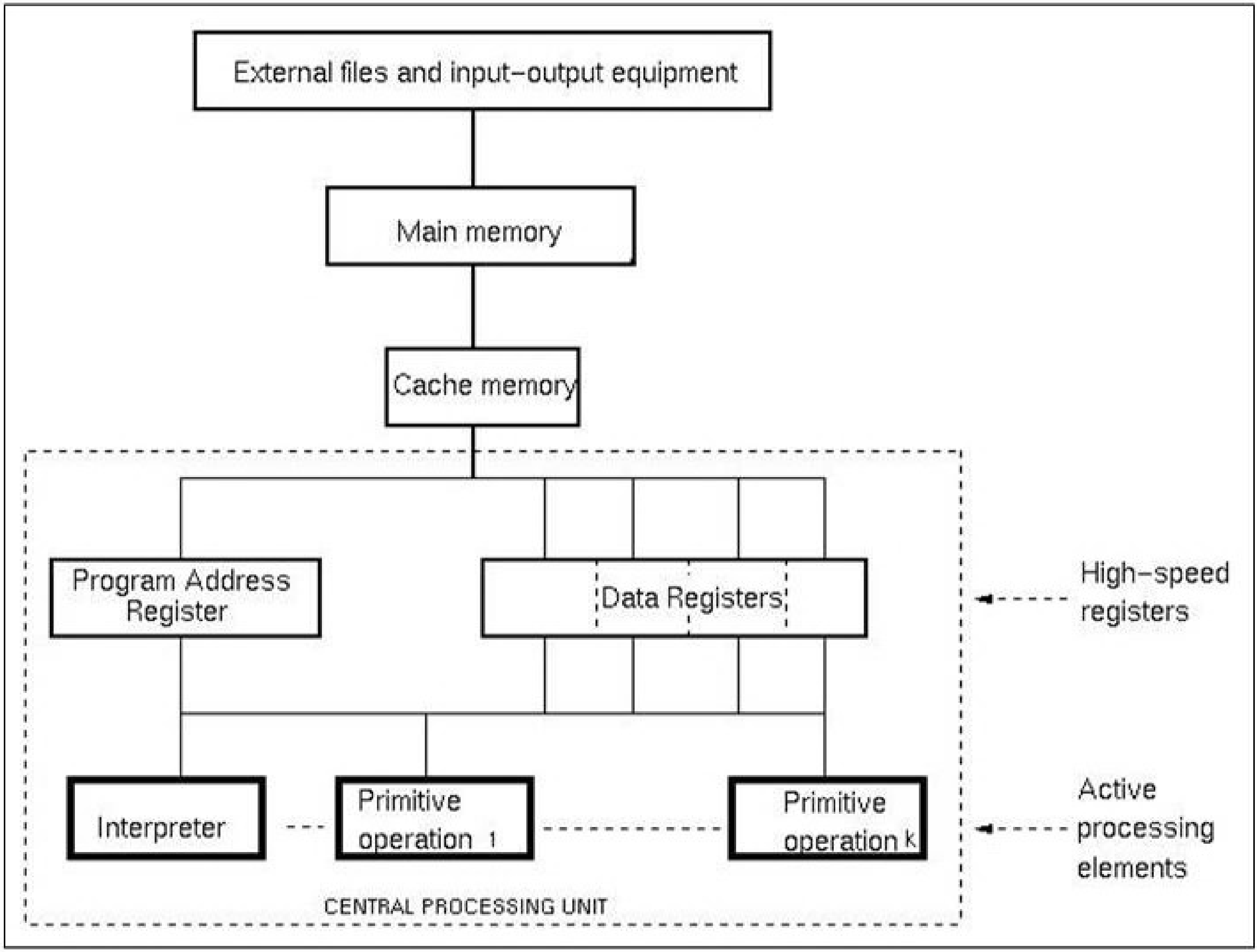
Tasks

A task differs little from the definition of an ordinary subprogram

- independent execution (thread of control)
- requires task synchronization and communication with other tasks - will look at communication later (semaphores)
- has separate address space for its own activation record

Processor design

Traditional processor design



Ways to speed up execution

To increase processor speed

Increase functionality of processor - add more complex instructions (CISC - Complex Instruction Set Computers)

Need more cycles to execute instruction:

2 cycles:

- fetch instruction
- execute instruction

How many:

- fetch instruction
- decode operands
- fetch operand registers
- decode instruction
- perform operation
- decode resulting address
- move data to store register
- store result

⇒ 8 cycles per instruction, not 2

Alternative - RISC

Have simple instructions, each executes in one cycle

- RISC - Reduced instruction set computer

Speed - one cycle for each operation, but more operations. For example: A=B+C

CISC: 3 instructions

Load Register 1, B
Add Register 1, C
Store Register 1, A

RISC:10 instruction

Address of B in read register
Read B
Move data Register 1
Address of C in read register
Read C
Move data Register 2
Add Register 1, Register 2, Register 3
Move Register 3 to write register
Address of A in write register
Write A to memory

Aspects of RISC design

Single cycle instructions

Large control memory - often more than 100 registers.

Fast procedure invocation - activation record invocation part of hardware. Put activation records totally within registers.

Implications

Cannot compare processor speeds of a RISC and CISC processor:

CISC - perhaps 8-10 cycles per instruction

RISC - 1 cycle per instruction

CISC can do some of these operations in parallel.

Pipeline architecture

CISC design:

1. Retrieve instruction from main memory.
2. Decode operation field, source data, and destination data.
3. Get source data for operation.
4. Perform operation.

Pipeline design:

while Instruction 1 is executing

Instruction 2 is retrieving source data

Instruction 3 is being decoded

Instruction 4 is being retrieved from memory.

Four instructions at once, with an instruction completion each cycle.

Impact on language design

With a standard CISC design, the statement E=A+B+C+D will have the postfix EAB+C+D+= and will execute as follows:

1. Add A to B, giving sum.
2. Add C to sum.
3. Add D to sum.
4. Store sum in E.

But, Instruction 2 cannot retrieve sum (the result of adding A to B until the previous instruction stores that result. This causes the processor to wait a cycle on Instruction 2 until Instruction 1 completes its execution.

A more intelligent translator would develop the postfix EAB+CD++=, which would allow A+B to be computed in parallel with C+D

Further optimization

```
   E=A+B+C+D
   J=F+G+H+I
has the postfix
   AB+FG+CD+HI+(1)(3)+(2)(4)+E(5)=F(6)=
```

[where the numbers indicate the operation number within that expression].

In this case, each statement executes with no interference from the other, and the processor executes at the full speed of the pipeline

Conditionals

```
A = B + C;
if D then E = 1
      ELSE E = 2
```

Consider the above program. A pipeline architecture may even start to execute (E=1) before it evaluates to see if D is true. Options could be:

- If branch is take, wait for pipeline to empty. This slows down machine considerably at each branch
- Simply execute the pipeline as is. The compiler has to make sure that if the branch is taken, there is nothing in the pipeline that can be affected.

This puts a great burden on the compiler writer. Paradoxically the above program can be compiled and run more efficiently as if the following was written:

```
if D (A=B+C) then E = 1
      ELSE E = 2
```

[Explain this strange behavior.]

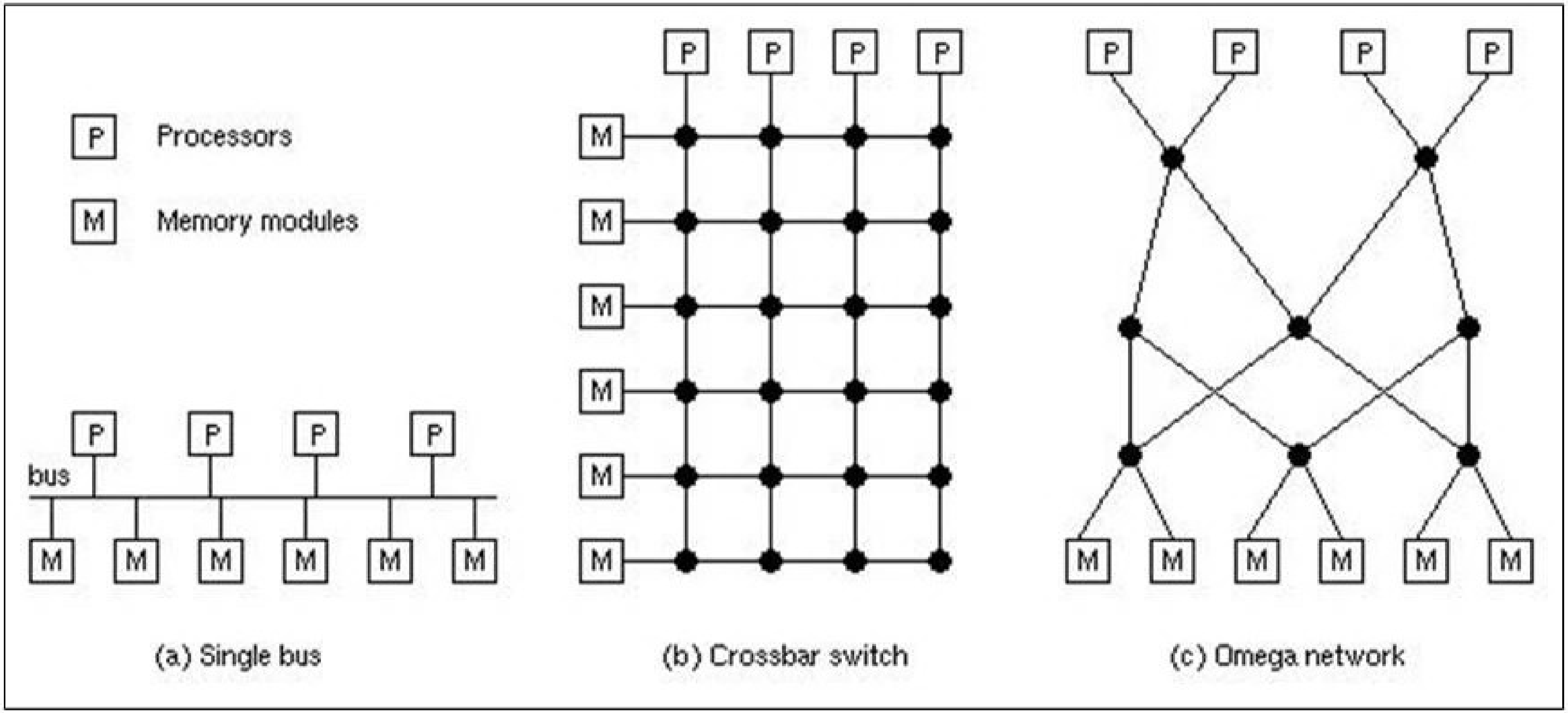
Summary

New processor designs are putting more emphasis on good language processor designs.

Need for effective translation models to allow languages to take advantage of faster hardware.

What’s worse - simple translation strategies of the past, may even fail now.

Multiprocessor system architecture



Impact: Multiple processors independently executing

Need for more synchronization

Cache coherence: Data in local cache may not be up to date.

Tightly coupled systems

Tightly coupled systems

- All processors have access to all memory
- Semaphores can be used to coordinate communication among processors
- Quick (nanosecond) access to all data

Loosely couples machines

- Processors have access to only local memory
- Semaphores cannot be used
- Relatively slow (millisecond) access to some data
- Model necessary for networks like the Internet