Hardware

INPUT DEVICE

transforms info into machine usable form

examples: terminal keyboard, magnetic disk unit, magnetic tape unit

OUTPUT DEVICE

transforms info from machine usable form into externally usable form

examples: line printer, terminal screen, magnetic disk unit, magnetic tape unit

CPU: "central processing unit"

the "brain" of the computer, manipulates the information and produces the results

CONTROL UNIT

instructs all the other parts what to do by following the commands of a program

coordinates and synchronizes:

CPU components and all other peripherals (I/O devices, external storage devices, etc.

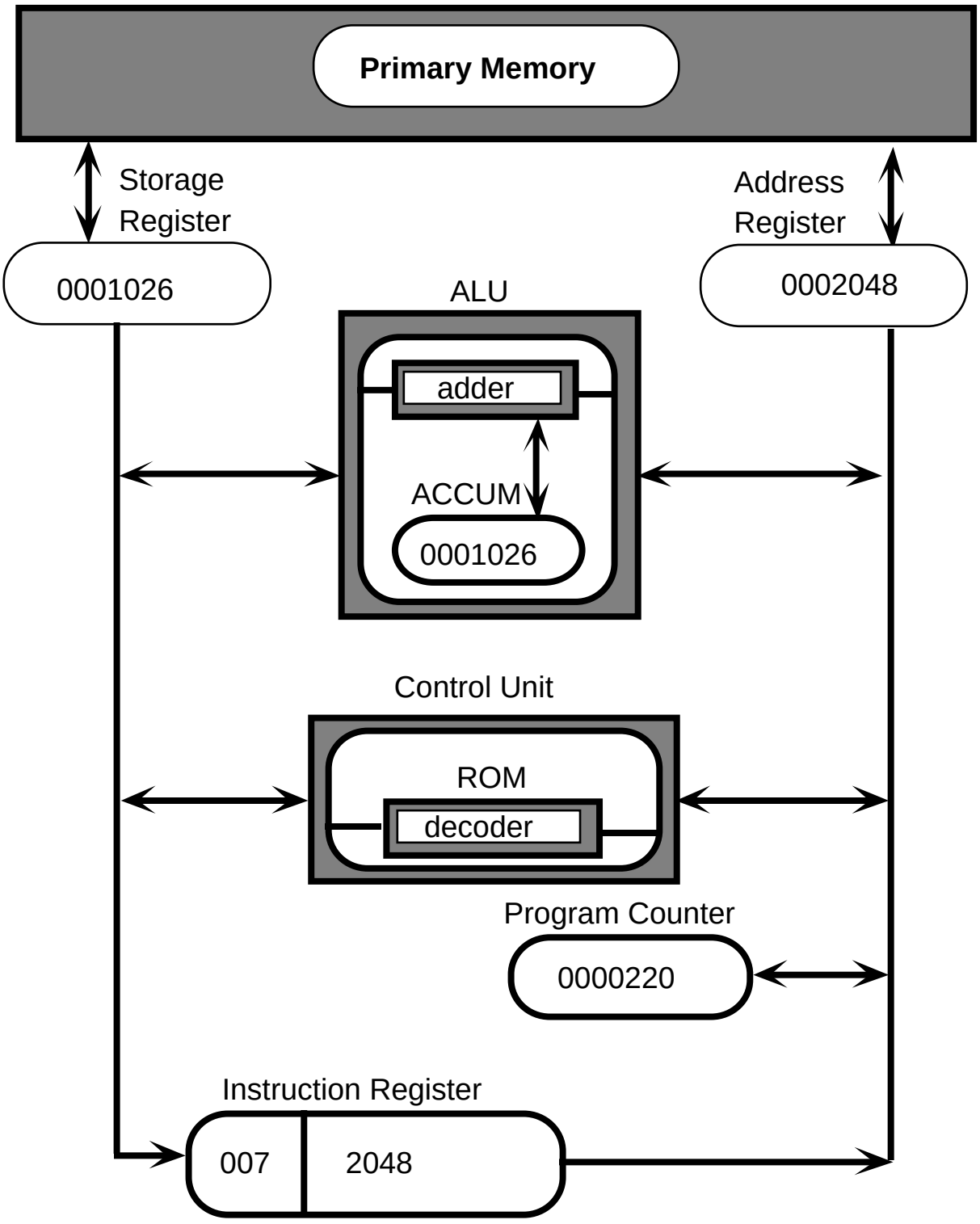
INTERNAL STORAGE (primary memory, main memory)

where program and data must reside before the program can execute (run); very fast, very expensive, relatively small

ALU "arithmetic logic unit"

performs all the arithmetic operations and the comparisons, etc; called the number cruncher

Computer pseudo-schematic



Software and Languages

Software is the collection of all the programs used to communicate with the computer; written in various languages

The computer can only understand one language, namely, its own machine language.

MACHINE LANGUAGE: (very low-level language)

- only language that the computer can understand
- consists of strings of 0's and 1's (bits)
- different for each machine
- all programs must be written in machine language (code) or
- be translated into machine code before being executed

ASSEMBLY LANGUAGE: (intermediate level)

- short abbreviations are used for instructions
- Fortran statement: X = Y + Z, might be translated:

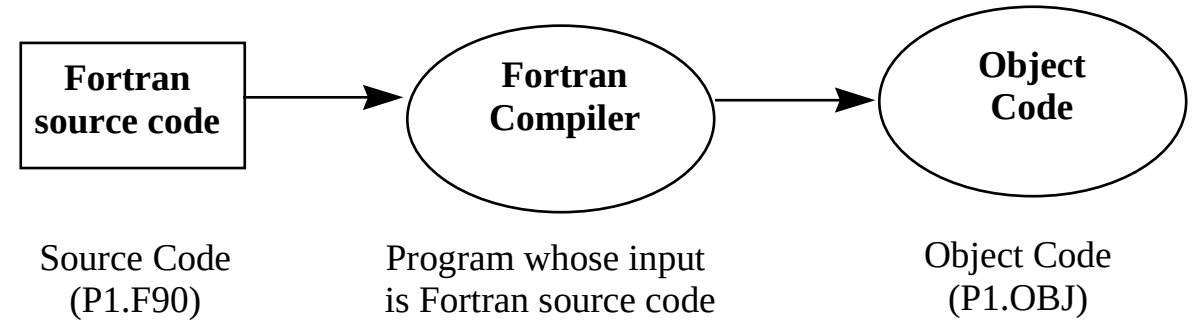
Machine code	Assembly language
10010011	LDA 9C2,A
00011010	ADD A,9C#
10000101	STO A,9C4
00110101	
10010001	
00010010	

Software and Languages

HIGH LEVEL LANGUAGES:

- •logical and English-like
- •machine independent
- •easy to read
- •must be translated (compiled or interpreted) before execution
- •Fortran (Formula translation), C, C++, Basic, Pascal, Cobol, PL1, etc.

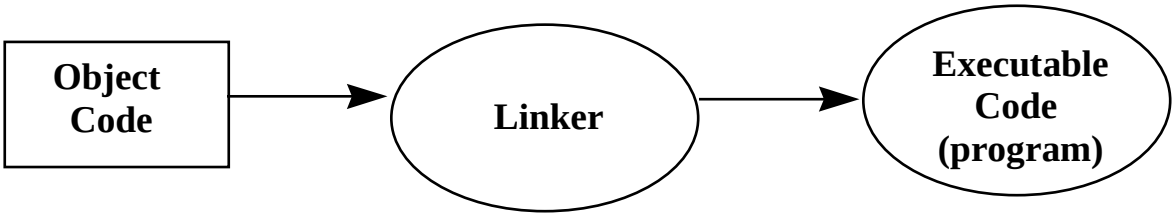
Compilation and linking:



Source Code (P1.F90)

Program whose input is Fortran source code

Object Code (P1.OBJ)



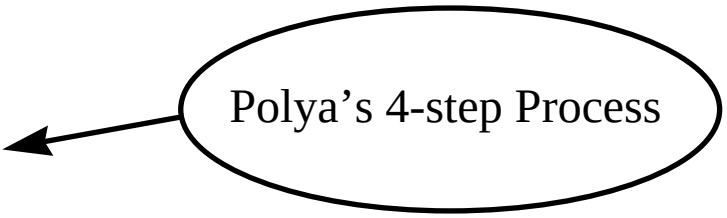
Object Code (P1.OBJ)

Program whose input is object code from a compiler

Executable Code (P1.EXE)

The Programming Process

Two-Phase Process



- Problem-solving phase
  - Analysis and Specification of the Problem
  - Creation of a General Solution (Algorithm)
  - Verification of the General Solution
- Implementation phase
  - Creation of a Specific Solution (Program)
  - Testing and Debugging of the Program

Of course, this is not generally a linear process. You will frequently have to go back and repeat or refine an earlier step in order to correct flaws or deal with unanticipated difficulties.

And even then, you're not finished:

Maintenance Phase

- Use of the Program (Install and Execute)
- Maintenance of the Program (Bug Fixes, Updates)

Polya's 4-step Process

1. Understand the problem
   a) Know the boundaries of the problem
   b) Know the constraints of the solution
   c) Know what actions are allowed

2. Devise a plan
   a) Organize thoughts to develop a detailed algorithm
   b) Use tools such as outlining, flowcharting and pseudocode to represent the plan in ever greater detail

3. Implement the plan
   a) Carry out the steps in the algorithm.
   b) Translate the problem into a language understood by the device to be used.

4. Test the plan
   a) Did the solution yield appropriate results?
   b) Can the solution be improved?

Understand the Problem

Questions to ask:

1. What is the required result?

2. What are the data?

3. What is the starting point (initial conditions)?

4. Is it at all possible to get the result from the data?

Devise a Plan

1. Have you seen a similar problem before?

2. Do you know of a related problem for which you know and understand a useful solution?
   Can you use part of the solution to the related problem?

3. Look at the data . . . logical repetitions in the data hint at loops in the solution.

4. If you can't solve the problem, can you solve part of it?

If you can do the problem by hand, consider:

1. What steps did you take?
2. Can you write them down clearly?
3. Is the problem divided into major logical parts?
4. Can you identify them clearly?
5. Have any problem assumptions been made?
6. If so, what are they?
7. Is the solution you've found general?

Implement the Plan

1. Check each step of the plan.

2. Have you considered all the special cases?

3. Can you arrive at a reasonable result given:
   a) reasonable data?
   b) unreasonable data?

4. Make certain your solution works for "boundary conditions":
   a) empty data
   b) data sets that are "too large"
   c) data sets that result in illegal operations such as division by zero

> Army Axiom: It is impossible to make anything foolproof, because fools are too ingenious.

Test the Plan

1. Did you compute any intermediate results that were not used later? If so those calculations can probably be eliminated.

2. Can the result be derived differently?

3. Can the solution be made simpler or more general?
   a) A simpler solution may yield a more efficient program.
   b) A more general solution may provide extra features or robustness.

4. Can the solution method be used for other problems?

Algorithm

What is an algorithm? One definition is:

"A finite set of instructions that specify a sequence of operations to be carried out in order to solve a specific problem or class of problems." [Zwass]

An algorithm must have the following properties:

- Finiteness: the algorithm must complete after a finite number of instructions have been executed.
- Absence of Ambiguity: each step must be clearly defined, having only one interpretation
- Definition of Sequence: each step must have a unique, defined preceding and succeeding step. The first step and last step must be clearly noted.
- Effectivenes: all instructions must be feasible. Illegal operations are not allowed.

Producing the Program

Polya's process yields a plan, not a program. The design phase of your solution is carried out (in theory, at least) with very little regard for the fact that you need to eventually produce a working program. In practice, the steps that are followed to turn an algorithm into a program often require that you return to the design phase and rethink or fine-tune the algorithm. In some cases you may even discover that is is necessary to abandon all the work done so far and start over. What follows is an attempt to outline, in a general way, the process of turning your algorithm into a working program:

1) Edit/enter the program:

writing down the symbols and words that make up the statements that compose the program (include comments !)

Legal Fortran Characters:

alphabetic	A - Z, a - z
numeric	0 - 9
alphanumeric	A - Z, a - z, 0 - 9
special symbols	β (blank char: NOT the letter (char) 'β')
	_ (underscore) = + - * / ( ) , . ‘ (single quote) : !
	“ (quote) % & ; < > ? $

Begin by writing code rather than by thinking and designing and you will take more time, discover unimagined nuances of frustration, and produce an inferior solution.

Producing the Program

2) Compile/Link the program. If you have errors, fix them.

3) Run the program — test it thoroughly. If you have errors, fix them.

Error detection — there are four general categories of errors:

A) Compiler (syntax) errors:

- error in the form of the statement: misspelled word, unmatched parenthesis, comma out of place, etc.
- detected by the compiler (at compile time)
- computer cannot correct error, so object program is not generated and thus program is not executed.
- computer (compiler) prints error messages, but continues to compile
- compiler error messages are often cryptic and terse; learning to recognize common ones will aid you greatly

B) Linking errors

- unusual for this course and programming environment (if things are set up correctly)
- most common cause is a misspelled function name or related error

Producing the Program

Error detection (continued):

C) Execution (runtime) errors

- program compiles OK, but something goes wrong during execution ... like division by 0, or an attempt to read data that does not exist, format error, etc.
- detected by the computer
- computer (usually) prints error message and stops !!

D) Logic errors

- program compiles, links and executes OK, but produces unexpected or incorrect results
- detected by programmer (YOU !!)
- hardest to find
  example: square_yard = length_yard + width_yard
- Suggestion: Hand trace the program.

Fortran Program Structure

Structure of a Fortran 90 Main Program

PROGRAM *name*

Declarations or specification statements

.

.

.

Executable statements

.

.

.

END PROGRAM *name*

- The main program must start with a PROGRAM statement
- The name in the PROGRAM statement must follow Fortran 90 naming conventions:
  - Characters allowed:A-Z, a-z, 0-9, and _ (underscore)
  - First character in a name must be a letter
  - A name must not contain more than 31 characters
- A program must end with an END statement

Fortran 90 Source Form

Constraints/Rules on Coding Programs in Fortran 90:

- A statement may begin anywhere on a line (free form)
- A line must not contain more than 132 characters
- Semicolons separate multiple statements on a line
- A trailing ampersand on a line indicates continuation of the statement to the next line
- A statement must not have more than 39 continuation lines
- Blanks separate keywords, labels, names, constants
- Statement labels range from 1 to 99999
- A comment on a line begins with an exclamation mark Compiler ignores all comments
- A statement may contain keywords (Fortran names) and identifiers (other names)
- An IMPLICIT NONE statement must immediately follow a PROGRAM statement for Essential Lahey Fortran 90
- Fortran is not a case-sensitive language