Computer Organization and Architecture, Pt. 2

Charles Abzug, Ph.D.
Department of Computer Science
James Madison University
Harrisonburg, VA 22807

Voice Phone: 540-568-8746, E-mail: CharlesAbzug@ACM.org
Home Page: http://www.cs.jmu.edu/users/abzugcx

© 2003 Charles Abzug

PROGRAMMING-LANGUAGE LEVELS

1. "High-Level" Languages: relatively machine-independent.

   Transportable from one machine architecture to another with at most minimal program changes, i.e., Platform-Independent.

   One HLL statement usually resolves to several machine-language instructions; relatively efficient.

   Relatively easily understandable/readable.

2. Machine Language and the associated Assembly Language: *highly* dependent upon the organization and architecture of the machine family.

   Transportable *only* within the same computer manufacturer's product line for the particular architecture selected , i.e., Platform-Dependent.

   Upwards compatible *only*, not downwards compatible.

   One Assembly-Language statement corresponds to one machine-language instruction; relatively speedy execution.

© 2003 Charles Abzug

Carpinelli Figure 3.1: COMPILATION and LINKAGE of PROGRAMS in a HIGH-LEVEL LANGUAGE

Original figure or table © 2001
by Addison Wesley Longman, Inc

Carpinelli Figure 3.2: ASSEMBLY and LINKAGE of ASSEMBLY-LANGUAGE PROGRAMS

**Original figure or table © 2001
by Addison Wesley Longman, Inc**

Carpinelli Figure 3.A:
COMPILATION and INTERPRETIVE EXECUTION of JAVA APPLETS

**Original figure or table © 2001
by Addison Wesley Longman, Inc**

STEPS in PROGRAM REALIZATION: (1) COMPILED LANGUAGE

1. Check source code for syntax errors; if any are found, then flag the errors, inform the programmer, and abort compilation.

2. If no errors, then continue.

3. Generate code:
   a) Optional intermediate step: *generate* ASL code.
   b) Generate object code (principally machine code).

4. Optimize object code.

5. Linkage Editor: link in any relevant library routines to produce executable code.

6. Load executable code into main memory.

7. Load *PC* with starting address of first instruction.

8. Run.

 © 2003 Charles Abzug

STEPS in PROGRAM REALIZATION: (2) ASSEMBLY LANGUAGE

1. Check source code for syntax errors; if any are found, then flag the errors, inform the programmer, and abort compilation.

2. If no errors, then continue.

3. First pass: *generate* symbol table, mark locations.

4. Second pass: *generate* machine code (object code).

5. Linkage Editor: link in any relevant library routines to produce executable code.

6. Load executable code into main memory.

7. Load *PC* with starting address of first instruction.

8. Run.

© 2003 Charles Abzug

STATEMENTS in ASSEMBLY LANGUAGE: Three Types

1. Machine Instructions:

2. Assembler Directives:

3. Macro Expansion Commands:

© 2003 Charles Abzug

STATEMENTS in ASSEMBLY LANGUAGE: Three Types

1. Machine Instructions:
   a) One line of assembly code corresponds to exactly one machine instruction.
   b) Assembled object code therefore contains one executable instruction per line of source code.
   c) Line may be rigidly formatted into fields.
   d) Mnemonics are designed for simultaneous simplicity and readability.
   e) Restrictions usually placed on label length and composition.

STATEMENTS in ASSEMBLY LANGUAGE: Three Types

1. Machine Instructions:
   a) One line of assembly code corresponds to exactly one machine instruction.
   b) Assembled object code therefore contains one executable instruction per line of source code.
   c) Line may be rigidly formatted into fields.
   d) Mnemonics are designed for simultaneous simplicity and readability.
   e) Restrictions usually placed on label length and composition.

2. Assembler Directives:
   a) Directive is executed at assembly time, usually resulting in some effect either upon the placement of part or all of the assembled code into memory or upon the content of initial data at program execution.
   b) Effect on run-time events is only indirect.

© 2003 Charles Abzug

STATEMENTS in ASSEMBLY LANGUAGE: Three Types

1. Machine Instructions:
   a) One line of assembly code corresponds to exactly one machine instruction.
   b) Assembled object code therefore contains one executable instruction per line of source code.
   c) Line may be rigidly formatted into fields.
   d) Mnemonics are designed for simultaneous simplicity and readability.
   e) Restrictions usually placed on label length and composition.

2. Assembler Directives:
   a) Directive is executed at assembly time, usually resulting in some effect either upon the placement of part or all of the assembled code into memory or upon the content of initial data at program execution.
   b) Effect on run-time events is only indirect.

3. Macro Expansion Commands:
   a) Purpose is to improve efficiency of the programmer.
   b) Enable a frequently-used sequence of instructions to be written once, but appear multiple times in the assembled program.
   c) Macro expansion is carried out at assembly time. Each line of macro expansion command usually results in multiple machine instructions in assembled program.

© 2003 Charles Abzug

EXECUTION of a MACHINE-LANGUAGE PROGRAM

1. Special-Purpose Registers: at least two in *EVERY* processor.

2. *INSTRUCTION REGISTER* (*IR*): holds the op-code of the currently-executing instruction.

3. *PROGRAM COUNTER* (*PC*): holds the memory address of the *NEXT* instruction to be executed (not the address of the instruction currently executing).

4. *MEMORY ADDRESS REGISTER* (*MAR*): contains an address whose content the processor needs either to write to or to read from.

5. *MEMORY BUFFER REGISTER* (*MBR*): contains a datum that the processor needs either to copy to the memory address specified in the *MAR* or that is being fetched from the address specified in the *MAR*.

© 2003 Charles Abzug

INSTRUCTION-EXECUTION CYCLE - Version 1

1. Fetch [i.e., copy the next instruction into the ***INSTRUCTION REGISTER***]. NOTE that the ***next instruction*** is defined to be the one located at the memory address whose value is specified in the special-purpose register called the ***PROGRAM COUNTER.***

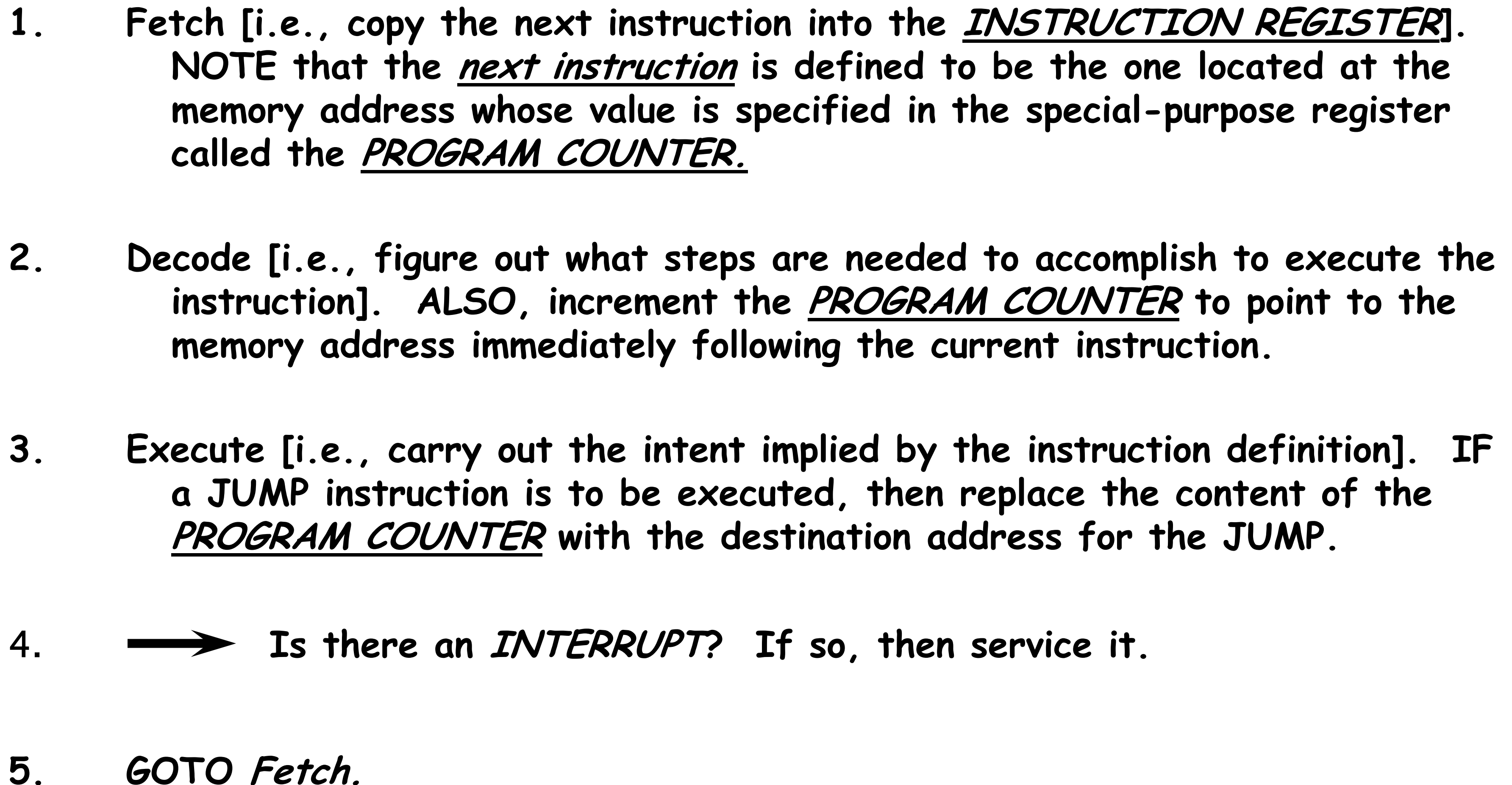
2. Decode [i.e., figure out what steps are needed to accomplish to execute the instruction]. *ALSO*, increment the ***PROGRAM COUNTER*** to point to the memory address immediately following the current instruction.

3. Execute [i.e., carry out the intent implied by the instruction definition]. IF a JUMP instruction is to be executed, then replace the content of the ***PROGRAM COUNTER*** with the destination address for the JUMP.

4. Repeat endlessly [i.e., *GOTO Fetch*].

© 2003 Charles Abzug

INSTRUCTION-EXECUTION CYCLE - Version 2



© 2003 Charles Abzug

TYPES of EXECUTABLE ASSEMBLY/MACHINE LANGUAGE INSTRUCTIONS

1. Data-Copy or Data-Transfer Instructions
   a) Load (copy from Main Memory into CPU, or from Input Device if I/O is memory-mapped).
   b) Store (copy from CPU into Main Memory, or into Input Device if I/O is memory-mapped).
   c) Move (copy within the CPU, or possibly between CPU and Main Memory).
   d) Input data from device to CPU (if I/O is not memory-mapped).
   e) Output data from CPU to device (if I/O is not memory-mapped).

2. Data-Operation Instructions
   a) Arithmetic instructions: Integer, Floating-Point, other.
   b) Logic instructions, including both bit-wise logical operations and shifts.

3. Program Control Instructions:
   a) Unconditional Jump Instruction.
   b) Conditional Jump Instructions: *JZ, JNZ, JN, JNN, JV, JNV, JC, JNC*
   c) Software Interrupts
   d) Exceptions & Traps

© 2003 Charles Abzug

DATA-COPY (DATA-TRANSFER) INSTRUCTIONS in Carpinelli's "Relatively Simple CPU"

Machine Code	Mnemonic	Description
0000 0001 Γ	LDAC	Load (i.e., copy into) the Accumulator with the contents of either a Main Memory location or data from an input device.
0000 0010 Γ	STAC	Store (i.e., copy the contents of) the Accumulator to Main Memory or to an output device.
0000 0011	MVAC	Move (i.e., copy) the contents of the Accumulator to Register *R*.
0000 0100	MOVR	Move (i.e., copy) the contents of Register *R* to the Accumulator.
0000 0000	NOP	*No OP*eration

© 2003 Charles Abzug

DATA-OPERATION INSTRUCTIONS in Carpinelli's "Relatively Simple CPU"

Machine Code	Mnemonic	Description
0000 1000	ADD	Add the contents of Register *R* to the current contents of the Accumulator, deposit the results in the Accumulator, and adjust the value of the *Z* bit.
0000 1001	SUB	Subtract the contents of Register *R* from the current contents of the Accumulator, deposit the results in the Accumulator, and adjust the value of the *Z* bit.
0000 1010	INAC	Increment the contents of the Accumulator, and adjust the value of the *Z* bit.
0000 1011	CLAC	Clear the contents of the Accumulator, and set the *Z* bit.
0000 1100	AND	Bitwise "AND" the contents of the Accumulator with the contents of Register *R*, and adjust the value of the *Z* bit.
0000 1101	OR	Bitwise "OR" the contents of the Accumulator with the contents of Register *R*, and adjust the value of the *Z* bit.
0000 1110	XOR	Bitwise "XOR" the contents of the Accumulator with the contents of Register *R*, and adjust the value of the *Z* bit.
0000 1111	NOT	Complement the contents of the Accumulator, and adjust the value of the *Z* bit.

© 2003 Charles Abzug

PROGRAM-CONTROL INSTRUCTIONS in Carpinelli's "Relatively Simple CPU"

Machine Code	Mnemonic	Description
0000 0101 Γ	JUMP	Instead of executing next the instruction following the current instruction, jump unconditionally to (i.e., execute next) the instruction situated at the specified memory location.
0000 0110 Γ	JMPZ	In the content of the *Z* register is a '1', then execute next the instruction situated at the specified memory location; otherwise, execute next the instruction immediately following the current instruction. Or, more simply, Jump on *Z*.
0000 0111 Γ	JPNZ	In the content of the *Z* register is a '0', then execute next the instruction situated at the specified memory location; otherwise, execute next the instruction immediately following the current instruction. Or, more simply, Jump on Not *Z*.

© 2003 Charles Abzug

Native Data Types

1. Integer and other Fixed-Point
    a) Binary:
        i. Non-Explicitly-Signed ("Unsigned")
        ii. Two's-Complement
        iii. Ones'-Complement
        iv. Signed-Magnitude
        v. Excess or Biased
    b) Decimal (BCD)

2. Floating-Point: (Sign of Mantissa), Mantissa, [Radix], Exponent
    USUALLY: Signed-Magnitude Mantissa, [Radix], Biased-or-Excess Exponent

3. Boolean

4. Single-Character & Character-String
    a) ASCII
    b) EBCDIC
    c) Unicode

© 2003 Charles Abzug

MODES of ADDRESSING

1. Direct Addressing Mode: Memory address is explicitly stated within the instruction, after the op code.
   EXAMPLE: *LDAC memory-address*, e.g.: *LDAC 5*
   EXECUTION: Copy into the Accumulator the content of memory address 5.

2. Indirect Addressing Mode: The memory address included within the instruction is <u>not</u> the address of the operand, but rather is the address *of the address* of the operand.
   EXAMPLE: *LDAC @address-of-operand's-memory-address*, e.g.: *LDAC @7*
   EXECUTION: Retrieve from address 7 a second address. Copy into the Accumulator the contents of the second address.

3. Register Direct Addressing Mode: The <u>value</u> of the operand is located within the specified register.
   EXAMPLE: *LDAC register-identifier*, e.g.: *LDAC R*
   EXECUTION: Copy into the Accumulator the contents of Register R.

4. Register Indirect Addressing Mode: The <u>memory address</u> of the operand is located within the specified register.
   EXAMPLE: *LDAC* (*register-identifier*), e.g.: *LDAC* (*R*), or
   LDAC @register-identifier, e.g.: *LDAC @R*
   EXECUTION: Retrieve from Register *R* a memory address. Copy into the Accumulator the contents of that address.

© 2003 Charles Abzug

MODES of ADDRESSING (continued)

5. Immediate Addressing Mode: The actual value of the operand is stated within the instruction.
   EXAMPLE: *LDAC #actual-value*, e.g.: *LDAC #3C*
   EXECUTION: Copies into the Accumulator the hex number *3C*.

6. Implicit Addressing Mode: The location of the operand is implied by the instruction itself, and can be inferred from the instruction mnemonic.
   EXAMPLE: *CLAC* (Clear the contents of the Accumulator).
   EXECUTION: Contents of the Accumulator changed to all zeroes.

7. Relative Addressing Mode: The numeric value specified within the instruction gives the <u>offset</u> of the desired memory location from the current contents of the Program Counter (NOTE: This is <u>*NOT*</u> the offset from the location of the currently-executing instruction, but rather the offset from the location of the <u>*NEXT*</u> instruction in sequence following the currently-executing instruction).
   EXAMPLE: *JMP $offset-amount*, e.g.: *JMP #3C*
   EXECUTION: If the *JMP* instruction starts at memory address *F000*, then the next instruction located after the *JMP* will be located at *F002*, and after execution of the *JMP*, the next instruction to be executed is at address *F03E* (= *F002* + *3C*)

© 2003 Charles Abzug

MODES of ADDRESSING (continued)

6. Indexed Addressing Mode: The numeric value specified within the instruction gives the <u>base address</u> of an array, while the contents of the Index Register indicate which array element is of current interest.
   EXAMPLE: *LDAC base-memory-location*(*X*), e.g.: *LDAC 102A*(*X*)
   EXECUTION: Assuming that the Index Register (Register *X*) contains the value *2005*, copy into the Accumulator the contents of memory address *302F* (= *102A* + *2005*)

7. Base Addressing Mode: A numeric value specified within a designated register (the Base Register) indicates a particular address, from which a numeric value indicated within the instruction gives the offset from the base address.
   EXAMPLE: *LDAC offset* (*register-containing-the-base-address*), e.g.: *LDAC 2005*(*102A*)
   EXECUTION: Copy into the Addumulator the contents of memory address *302F* (= the sum of the base memory location *102A* and the offset *2005*)

© 2003 Charles Abzug

Carpinelli Figure 3.3, part 1, from the text but enhanced: GENERATION of ADDRESSES for Various MODES of ADDRESSING, part 1

Direct addressing mode:

Indirect addressing mode:

Register Direct addressing mode:

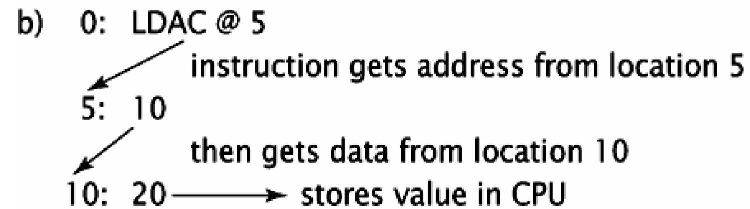
Register Indirect addressing mode:

Original figure or table © 2001 by Addison Wesley Longman, Inc

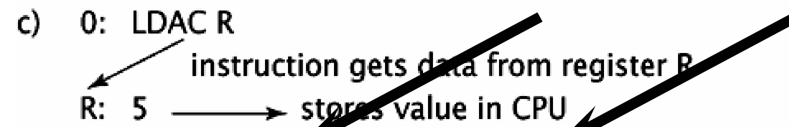
Carpinelli Figure 3.3, part 1, *ENHANCED & CORRECTED*: GENERATION of ADDRESSES for Various MODES of ADDRESSING, part 1

Direct addressing mode:

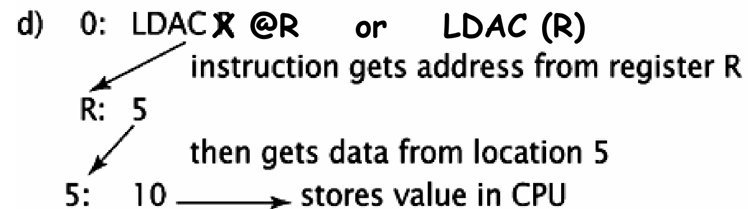
Indirect addressing mode:



Register Direct addressing mode:



Register Indirect addressing mode:



Original figure or table © 2001 by Addison Wesley Longman, Inc

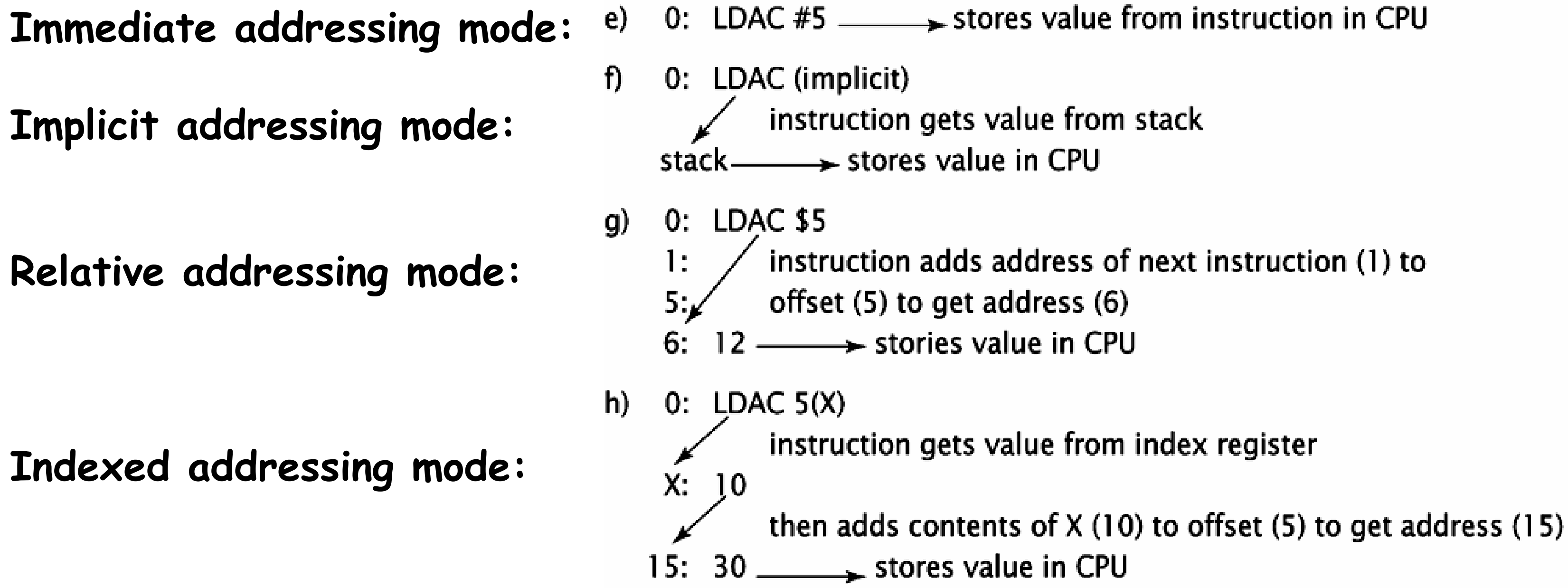
Carpinelli Figure 3.3, part 2, from the text but enhanced: GENERATION of ADDRESSES for Various MODES of ADDRESSING, part 2

Immediate addressing mode:

Implicit addressing mode:

Relative addressing mode:

Indexed addressing mode:



Original figure or table © 2001 by Addison Wesley Longman, Inc

Carpinelli Figure 3.3, part 2, *ENHANCED & IMPROVED*: GENERATION of ADDRESSES for Various MODES of ADDRESSING, part 2

Immediate addressing mode:

e) 0: LDAC #5 → stores value from instruction in CPU

Implicit addressing mode:

f) 0: LDAC (implicit)
instruction gets value from stack
stack → stores value in CPU

Relative addressing mode:

g) 0: JNZ $5
Instruction adds the address of the <u>next</u> instruction (2)
2: to the stated offset (5) to get the destination address (7) for the jump. The number 7 is loaded into the *PC*.

Indexed addressing mode:

h) 0: LDAC 5(X)
instruction gets value from index register
X: 10
then adds contents of X (10) to offset (5) to get address (15)
15: 30 → stores value in CPU

Original figure or table © 2001 by Addison Wesley Longman, Inc

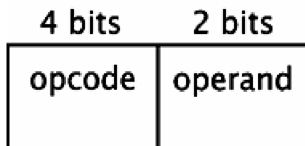
Carpinelli Figure 3.4: INSTRUCTION CODE FORMATS, ASSEMBLY LANGUAGE, and MACHINE CODE

Original figure or table © 2001 by Addison Wesley Longman, Inc

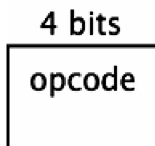
Carpinelli Figure 3.4: INSTRUCTION CODE FORMATS, ASSEMBLY LANGUAGE, and MACHINE CODE

Assembly	Operation	Machine code
ADD *A,B,C*	(*A*=*B*+*C*)	1010 00 01 10

Assembly	Operation	Machine code
MOVE *A,B*	(*A*=*B*)	1000 00 01
ADD *A,C*	(*A*=*A*+*C*)	1010 00 10



Assembly	Operation	Machine code
LOAD *B*	(*Acc*=*B*)	0000 01
ADD *C*	(*Acc*=*Acc*+*C*)	1010 10
STORE *A*	(*A*=*Acc*)	0001 00



Assembly	Operation	Machine code
PUSH *B*	(Stack=*B*)	0101
PUSH *C*	(Stack=*C,B*)	0110
ADD	(Stack=*B*+*C*)	1010
POP *A*	(*A*=stack)	1100

NOTE: Normal practice is to distinguish different versions of the same machine instruction having different addressing modes <u>either</u> by using different op codes <u>or</u> by designating a bit field within the instruction format as the mode field.

Original figure or table © 2001 by Addison Wesley Longman, Inc

ELEMENTS of INSTRUCTION SET ARCHITECTURE

1. Registers: types, size or width for each type, number present of each type, ASL names.

2. Machine Instructions: actions/effects, op-codes, ASL mnemonics, number of operands (source plus destination) for each instruction.

3. Addressing modes, and the standards for their specification in ASL as well as in machine language.

4. Procedures required for enablement and disablement of interrupts.

5. Flags: special-purpose one-bit registers.
   a) Status flags: indicate the status of the latest operation:
      Zero, Negative, Carry, oVerflow: set or cleared automatically by the CPU.
   b) Parity flag: set or cleared automatically by the CPU.
   c) Interrupt Mask (Interrupt-enabled/disabled flag): set or cleared via explicit command in program.

© 2003 Charles Abzug

ISSUES in INSTRUCTION-SET ARCHITECTURE

1. Completeness of the instruction set: Are all necessary operations included?

2. Orthogonality of the instruction set: little or no overlap of functionality between instructions.

3. Numbers and Types of registers:
   a) Integer/Fixed-Point
   b) Floating-Point
   c) BCD or other special-purpose/use
   d) Multi-Use Registers

© 2003 Charles Abzug

Carpinelli Table 3.1: INSTRUCTION SET for the "RELATIVELY SIMPLE CPU"

Original figure or table © 2001 by Addison Wesley Longman, Inc

Carpinelli Figure 3.5, *ENHANCED*: INSTRUCTION FORMATS for the "RELATIVELY SIMPLE CPU"

One-Byte Instruction:

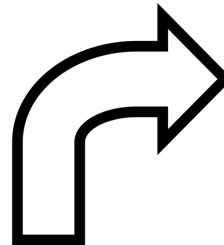
Three-Byte Instruction:

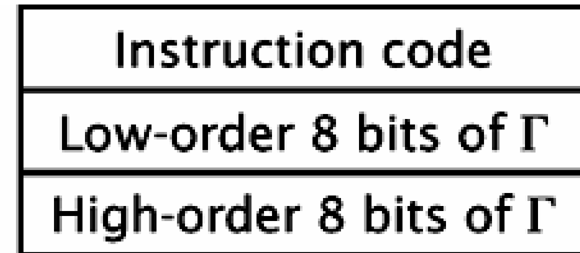
Original figure or table © 2001 by Addison Wesley Longman, Inc:

Carpinelli Figure 3.5, *ENHANCED and IMPROVED*: INSTRUCTION FORMATS for the "RELATIVELY SIMPLE CPU"

One-Byte Instruction:

Three-Byte Instruction:





NOTE that the order of storage of the two bytes of the memory address shown here corresponds to LITTLE-ENDIAN notation.

Original figure or table © 2001 by Addison Wesley Longman, Inc:

LOOP-SUMMATION PROGRAM for the “RELATIVELY SIMPLE CPU”)

```
        CLAC
        STAC total      % Initialize total to zero.
        STAC i          % Initialize loop counter to zero.

Loop:   LDAC i
        INAC
        STAC i          % Increment the stored value of the loop counter.

        MVAC            % Copy the loop counter to Register R.
        LDAC total
        ADD             % Add the value of the loop counter to the total.
        STAC total

        LDAC n
        SUB
        JPNZ Loop       % Go to the top of the loop, unless i = n.

        HALT
```


© 2003 Charles Abzug

IMPLEMENTATION of a HIGH-LEVEL-LANGUAGE STATEMENT in ASSEMBLY LANGUAGE:
X = A + (B*C) + D

1. Three-Operand Instructions available on machine:

```
        XOR     R1, R1, R1      % Create in count register (R1) a content of 0.
        MOV     R2, R1          % Copy the 0 to calculation end-result register
                                          (R2).
        LOAD    R3, B           % Copy the multiplicand to Register R3
        LOAD    R4, C           % Copy the multiplier to Register R4.

Loop:   INCR    R1              % Increment counter.
        ADD     R2, R2, R3      % Add value of multiplicand to end-result.
        CMP     R5, R1, R4      % Compare count to multiplier, discard the result.
        JNZ     Loop:           % Go back and add multiplicand another time.

                                % Finished multiplying; Register R2 contains B*C.
        LOAD    R3, A           % Bring the value of A into the CPU.
        ADD     R2, R2, R3      % Add the value of A to the product B*C.
        LOAD    R3, D           % Bring the value of D into the CPU.
        ADD     R2, R2, R3      % Add the value of D into the cumulative sum.
        STOR    X, R2           % Copy out the final answer.
```


© 2003 Charles Abzug

IMPLEMENTATION of a HIGH-LEVEL-LANGUAGE STATEMENT in ASSEMBLY LANGUAGE: X = A + (B*C) + D

2. Two-Operand Instructions (but not three-operand) available on machine:

```
        XOR   R1, R1      % Create in count register (R1) a content of 0.
        MOV   R2, R1      % Copy the 0 to calculation end-result register
                                  (R2).
        LOAD  R3, B       % Copy the multiplicand to Register R3
        LOAD  R4, C       % Copy the multiplier to Register R4.

Loop:   INCR  R1          % Increment counter.
        ADD   R2, R3      % Add value of multiplicand to end-result.
        CMP   R4, R1      % Compare count to multiplier.
        LOAD  R4, C       % Restore the multiplier to Register R4.
        JNZ   Loop:       % Go back and add multiplicand another time.
                          % Finished multiplying; Register R2 contains B*C.
        LOAD  R3, A       % Bring the value of A into the CPU.
        ADD   R2, R3      % Add the value of A to the product B*C.
        LOAD  R3, D       % Bring the value of D into the CPU.
        ADD   R2, R3      % Add the value of D into the cumulative sum.
        STOR  X, R2       % Copy out the final answer.
```


© 2003 Charles Abzug

IMPLEMENTATION of a HIGH-LEVEL-LANGUAGE STATEMENT in ASSEMBLY LANGUAGE:
X = A + (B*C) + D

3. One-Operand Instructions (but not two- or three-operand) available on machine:

```
        CLAC            % Create in the Accumulator a content of 0.
        STAC    X       % Initialize the value of final result to 0.
        STAC    Count   % Initialize the value of the counter to 0.

Loop:   LDAC    B       % Copy the value of the multiplicand to the Accumulator.
        MVAC            % Copy the multiplicand to Register R.
        LDAC    X       % Load current value of end-result into the Accumulator.
        ADD             % Add value of multiplicand to end-result.
        STAC    X       % Copy out the current value of end-result.
        LDAC    Count   % Prepare to update count.
        INAC            % Update the count.
        STAC    Count   % Store the updated count.
        MVAC            % Copy the updated count to Register R.
        LDAC    C       % Load the value of the multiplier.
        SUB             % Compare the current count to the multiplier.
        JNZ     Loop    % Continue multiplying.


                        % (continued)
```


© 2003 Charles Abzug

IMPLEMENTATION of a HIGH-LEVEL-LANGUAGE STATEMENT in ASSEMBLY LANGUAGE: X = A + (B*C) + D

```
                %               Continuation of ONE-Operand ASL code:
                        % Finished multiplying; X contains B*C.
LDAC    A               % Bring the value of A into the CPU.
MVAC                    % Copy the value of A to Register R.
LDAC    X               % Copy the current value of end-result (= B*C) into the
                        %       Accumulator.
ADD                     % Add to B*C the value of A.
STAC    X               % Copy back to memory the current value of end-result
                        %       (= A + B*C).
LDAC    D               % Bring the value of D into the CPU.
MVAC                    % Copy the value of D to Register R.
LDAC    X               % Copy the current value of end-result (= A + B*C) into
                        %        the Accumulator.
ADD                     % Add to A + B*C the value of D.
STAC    X               % Copy out the final answer (= A + B*C + D).
```


© 2003 Charles Abzug

IMPLEMENTATION of a HIGH-LEVEL-LANGUAGE STATEMENT in ASSEMBLY LANGUAGE:
X = A + (B*C) + D

4. No-Operand Instructions (but not one, two- or three-operand) available:

```
        CLAC                % Create in the Accumulator a content of 0.
        PUSHAC
        POP      X          % Initialize the value of final result to 0.
        PUSHAC
        POP      Count      % Initialize the value of the counter to 0.

Loop:   PUSH     B          % Copy the value of the multiplicand to the Stack.
        PUSH     X          % Load current value of end-result onto the .
        ADD                 % Add value of multiplicand to end-result.
        POP      X          % Copy out the current value of end-result.
        PUSH     Count      % Prepare to update count.
        PUSH     #1
        ADD                 % Update the count.
        POP      Count      % Store the updated value of count.
        PUSH     Count      % Re-copy the current value of count to the stack.
        PUSH     C          % Load the value of the multiplier.
        SUB                 % Compare the current count to the multiplier.


                            % (continued)
```


© 2003 Charles Abzug

IMPLEMENTATION of a HIGH-LEVEL-LANGUAGE STATEMENT in ASSEMBLY LANGUAGE: X = A + (B*C) + D

```
                %           Continuation of NO-Operand ASL code:

POP     Discard    % Remove from stack and discard the difference between
                   %          multiplier and count.

JNZ     Loop       % Continue multiplying.

                   % Finished multiplying; X contains B*C.
PUSH    A          % Copy the value of A onto the Stack.
PUSH    X          % Copy the current value of end-result (= B*C) onto the
                   %          Stack.
ADD                % Add to B*C the value of A.
PUSH    D          % Copy the value of D onto the Stack.
ADD                % Add to A + B*C the value of D.
POP     X          % Copy out the final answer (= A + B*C + D).
```


© 2003 Charles Abzug

Carpinelli Table 3.2: EXECUTION TRACE for the LOOP SUMMATION PROGRAM

Original figure or table © 2001 by Addison Wesley Longman, Inc

Carpinelli Table 3.3: DATA MOVEMENT (COPY) INSTRUCTIONS for the *Intel* 8085 MICROPROCESSOR

Original figure or table © 2001 by Addison Wesley Longman, Inc

Carpinelli Figure 3.6: INSTRUCTION FORMATS for the *Intel* 8085 MICROPROCESSOR

Original figure or table © 2001
by Addison Wesley Longman, Inc

Carpinelli Table 3.4: DATA OPERATION INSTRUCTIONS for the *Intel* 8085 MICROPROCESSOR

Original figure or table © 2001 by Addison Wesley Longman, Inc

Carpinelli Table 3.5: PROGRAM CONTROL INSTRUCTIONS for the *Intel* 8085 MICROPROCESSOR

Original figure or table © 2001 by Addison Wesley Longman, Inc

Carpinelli Table 3.6:
EXECUTION TRACE
of the LOOP SUMMATION PROGRAM
for the *Intel* 8085 MICROPROCESSOR

Original figure or table © 2001 by Addison Wesley Longman, Inc

Overview of Computer Organization

© 2003 Charles Abzug

Carpinelli Figure 4.1: GENERIC COMPUTER ORGANIZATION

Original figure or table © 2001
by Addison Wesley Longman, Inc

What Is a BUS?

1. ***Physical Bus***: A set of wires operating as a unit for the purpose of conveying data of some type (instructions or program data) between different functional subunits of the computer.

2. ***Bus Protocol***: A precisely defined set of rules governing how data are transmitted over the bus: *Who* does *What,* and in *What Order*.

© 2003 Charles Abzug

MAJOR COMPONENTS of a BUS

1. Address Lines
2. Data Lines
3. Control Lines
4. Power Lines

© 2003 Charles Abzug

Types of BUSES

1. Main Bus or System Bus:
   a) connects Main Memory to the CPU
   b) usually the fastest bus in the system

2. Auxiliary or Local or I/O Bus:
   a) connects I/O devices, usually slower than Main Memory

© 2003 Charles Abzug

TYPES of MEMORY

1. Read/Write Memory (conventionally known as ***RAM***, or "Random Access Memory", a name that distinguishes this kind of memory from "Sequential Access Memory" and from "Pseudo-Random Access Memory")

2. Write-with-Difficulty-but-Read-with-Ease Memory (conventionally known as ***ROM***, or "Read-Only Memory").
   a) In its original form, the name was accurate: content was burned in at chip fabrication.
   b) Second form was ***PROM***, or "Programmable ROM". Generic chips produced, could be programmed individually.
   c) ***EPROM***, or "Erasable PROM": erased with UV light, then could be reprogrammed.
   d) ***EEPROM***, or "Electrically Erasable PROM": much more readily reprogrammed than EPROM.

© 2003 Charles Abzug

Carpinelli Figure 4.2:
TIMING DIAGRAMS for MEMORY BUS OPERATIONS

Original figure or table © 2001
by Addison Wesley Longman, Inc

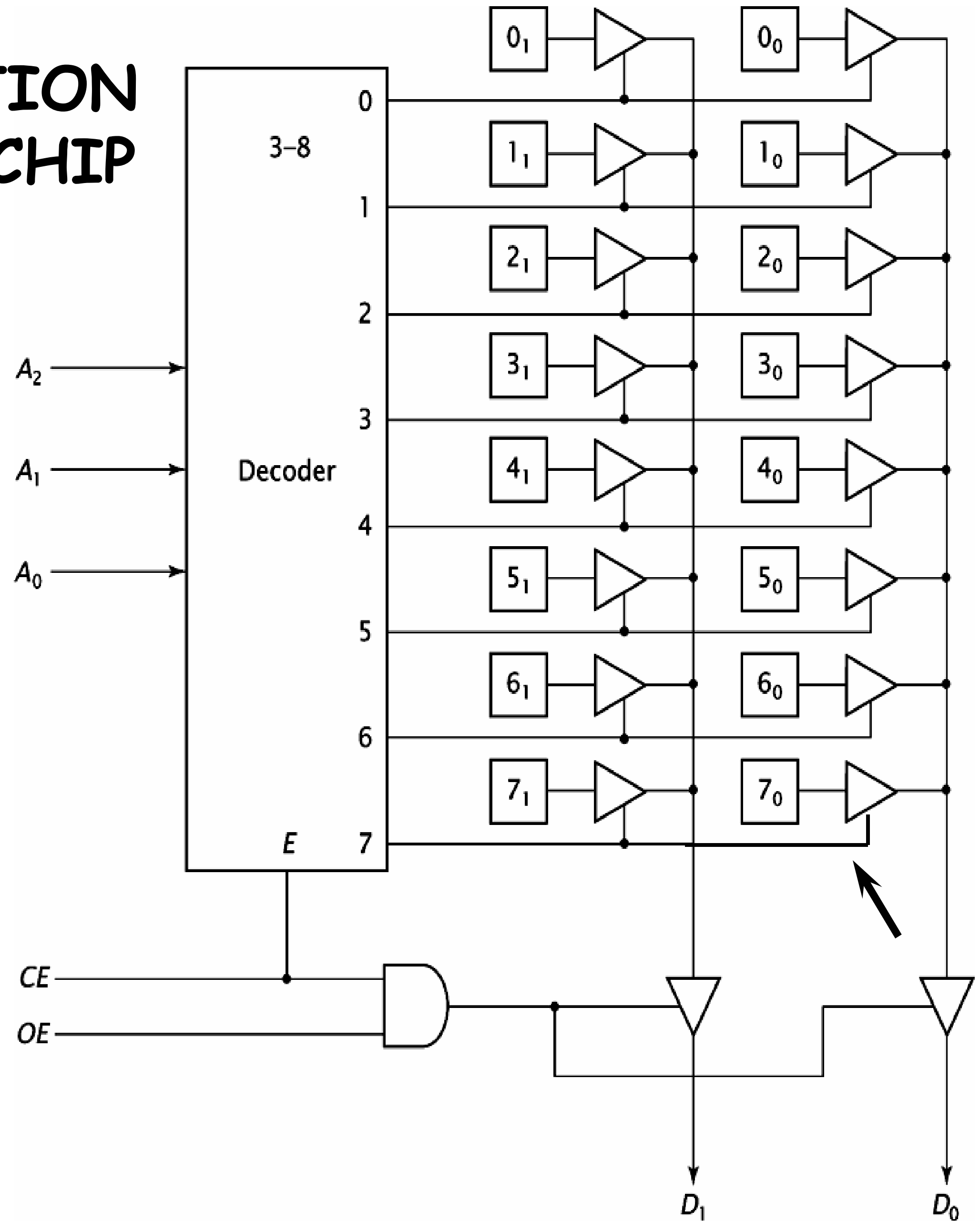
Carpinelli Figure 4.3:
INTERNAL ORGANIZATION of the CPU

Original figure or table © 2001
by Addison Wesley Longman, Inc

Carpinelli Figure 4.4, as it appears in the text: INTERNAL ORGANIZATION of a LINEAR 8x2 ROM CHIP

Original figure or table © 2001
by Addison Wesley Longman, Inc

Carpinelli Figure 4.4, *CORRECTED*: INTERNAL ORGANIZATION of a LINEAR 8x2 ROM CHIP



Original figure or table © 2001
by Addison Wesley Longman, Inc

Carpinelli Figure 4.5:
INTERNAL TWO-DIMENSIONAL ORGANIZATION of an 8x2 ROM CHIP

**Original figure or table © 2001
by Addison Wesley Longman, Inc**

Carpinelli Figure 4.6: CONSTRUCTION of an 8x4 MEMORY SUBSYSTEM from TWO 8x2 ROM CHIPS

Original figure or table © 2001
by Addison Wesley Longman, Inc

Carpinelli Figure 4.8:
CONSTRUCTION of an 8x4 MEMORY SUBSYSTEM from TWO 8x2 ROM CHIPS, *CONTROL SIGNALS ADDED*

Original figure or table © 2001
by Addison Wesley Longman, Inc

Carpinelli Figure 4.7 (a): CONSTRUCTION of a 16x2 MEMORY SUBSYSTEM from TWO 8x2 ROM CHIPS, *SEQUENTIAL ADDRESSING*

Original figure or table © 2001
by Addison Wesley Longman, Inc

Carpinelli Figure 4.7 (b): CONSTRUCTION of a 16x2 MEMORY SUBSYSTEM from TWO 8x2 ROM CHIPS, *INTERLEAVED ADDRESSING*

**Original figure or table © 2001
by Addison Wesley Longman, Inc**

Memory Subsystem Architecture

1. Harvard Architecture: separate memory modules for storage of data and of instructions.

2. "von" Neumann Architecture: data and instructions intermingled in a single memory module.

© 2003 Charles Abzug

Carpinelli Table 4.1: ALTERNATIVE CONVENTIONS for REPRESENTION of MULTIPLE-BYTE DATA

Original figure or table © 2001
by Addison Wesley Longman, Inc

Memory Address	Data (in hex)
100	01
101	02
102	03
103	04

Big-Endian

Memory Address	Data (in hex)
100	04
101	03
102	02
103	01

Little-Endian

Carpinelli Figure 4.9: An INPUT DEVICE

Original figure or table © 2001
by Addison Wesley Longman, Inc

Carpinelli Figure 4.10: An OUTPUT DEVICE

Original figure or table © 2001
by Addison Wesley Longman, Inc

Carpinelli Figure 4.11:
A BIDIRECTIONAL INPUT/OUTPUT DEVICE

**Original figure or table © 2001
by Addison Wesley Longman, Inc**

Carpinelli Figure 4.12: CPU DETAILS for Carpinelli's "RELATIVELY-SIMPLE COMPUTER"

Original figure or table © 2001
by Addison Wesley Longman, Inc

Carpinelli Figure 4.13: MEMORY SUBSYSTEM DETAILS for Carpinelli's "RELATIVELY-SIMPLE COMPUTER"

Original figure or table © 2001
by Addison Wesley Longman, Inc

Carpinelli Figure 4.14: FINAL DESIGN for Carpinelli's "RELATIVELY-SIMPLE COMPUTER"

Original figure or table © 2001
by Addison Wesley Longman, Inc

Carpinelli Figure 4.15: DEMULTIPLEXING of the *AD* PINS for the *Intel* 8085 MICROPROCESSOR

Original figure or table © 2001
by Addison Wesley Longman, Inc

Original figure © 1979
by Intel Corporation

Carpinelli Figure 4.16: A Minimal 8085-based Computer System

NOTE 1: TRAP. INTR. AND HOLD MUST BE GROUNDED IF THEY AREN'T USED.

NOTE 2: USE 10/M FOR STANDARD 1/0 MAPPING USE A15 FOR MEMORY MAPPED I/O.

NOTE 3: CONNECTION IS NECESSARY ONLY IF ONE T_{WAIT} STATE IS DESIRED.

NOTE 4: PULL-UP RESISTORS RECOMMENDED TO AVOID SPURIOUS SELECTION WHEN RD AND WR ARE 3-STATED. THESE RESISTORS ARE NOT INCLUDED ON THE PC BOARD LAYOUT OF FIGURE 3-7.

Original figure © 1979
by Intel Corporation

Carpinelli Figure 4.16: A Minimal 8085-based Computer System: MAGNIFIED PARTIAL DETAIL

Original figure © 1979
by Intel Corporation

Carpinelli Figure 4.16: A Minimal 8085-based Computer System: MAGNIFIED PARTIAL DETAIL

Register Transfer Language (RTL)

and

Micro-Operations (Micro-Ops)

© 2003 Charles Abzug

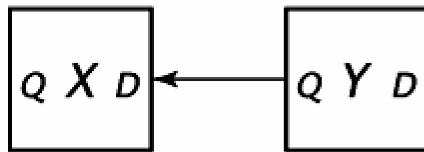
Carpinelli Figure 5.1, as it appears in the text: IMPLEMENTATION of the MICRO-OPERATION $X \leftarrow Y$

Direct
Connection

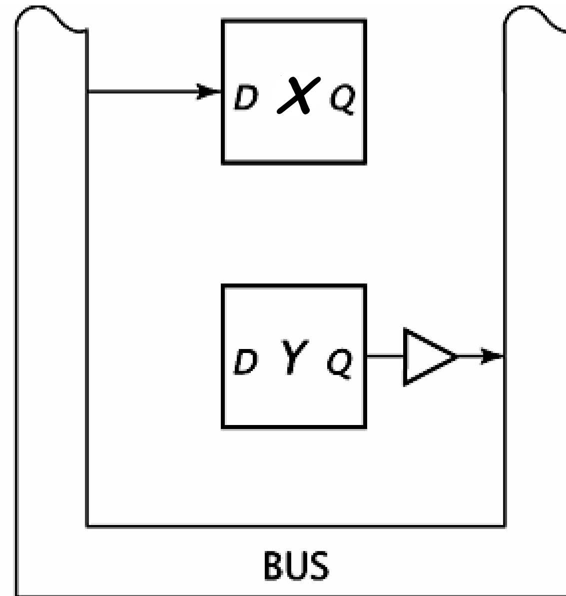
Bus
Connection

Original figure or table © 2001
by Addison Wesley Longman, Inc

Carpinelli Figure 5.1, *CORRECTED*: IMPLEMENTATION of the MICRO-OPERATION $X \leftarrow Y$



Direct Connection



Bus Connection

Original figure or table © 2001
by Addison Wesley Longman, Inc

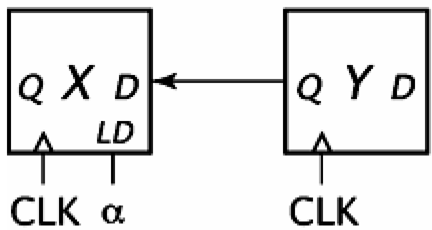
Carpinelli Figure 5.2, as it appears in the text: IMPLEMENTATION of the DATA TRANSFER α:$X \leftarrow Y$

Direct Path

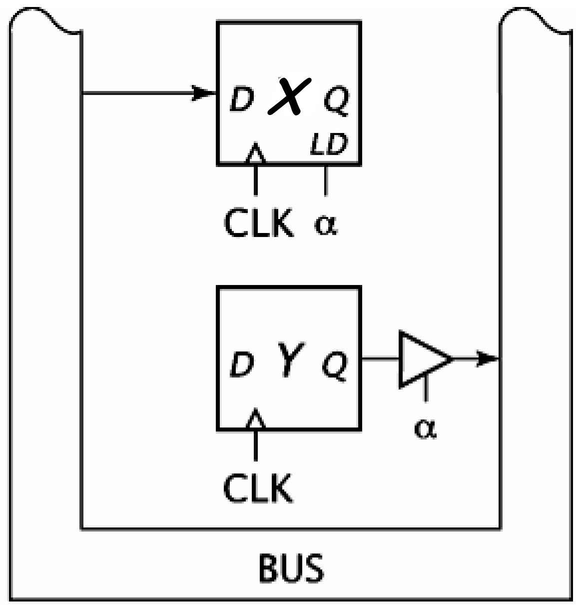
Bus

Original figure or table © 2001
by Addison Wesley Longman, Inc

Carpinelli Figure 5.2, *CORRECTED*: IMPLEMENTATION of the DATA TRANSFER α:X←Y



Direct Path



Bus

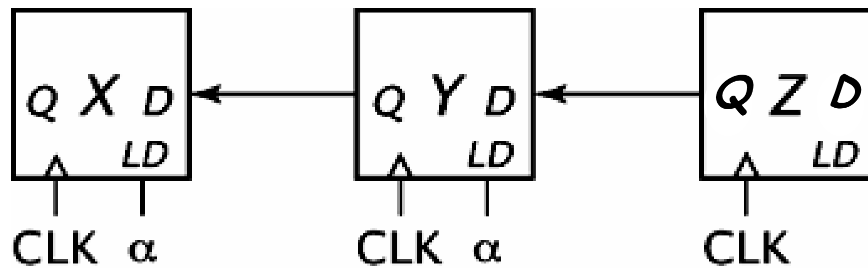
Original figure or table © 2001
by Addison Wesley Longman, Inc

Carpinelli Figure 5.3, as it appears in the text: IMPLEMENTATION of the DATA TRANSFER $\alpha: X \leftarrow Y,\ Y \leftarrow Z$

NOTE: There must be two independent data-transfer paths available.

Original figure or table © 2001
by Addison Wesley Longman, Inc

Carpinelli Figure 5.3, *CORRECTED*: IMPLEMENTATION of the DATA TRANSFER α:X←Y, Y←Z



NOTE: There must be two independent data-transfer paths available.

Original figure or table © 2001
by Addison Wesley Longman, Inc

Carpinelli Figure 5.4: IMPLEMENTATION of the DATA TRANSFER α: $X \leftarrow Y$, $Z \leftarrow Y$

Original figure or table © 2001
by Addison Wesley Longman, Inc

Carpinelli Figure 5.5: IMPLEMENTATION of the DATA TRANSFER α:X←0, β:X←1

Original figure or table © 2001
by Addison Wesley Longman, Inc

Carpinelli Figure 5.6:
IMPLEMENTATION of the FOUR-BIT DATA TRANSFER
α: $X \leftarrow Y$

Original figure or table © 2001
by Addison Wesley Longman, Inc

Carpinelli Table 5.1: ARITHMETIC and LOGICAL MICRO-OPERATIONS

Original figure or table © 2001
by Addison Wesley Longman, Inc

Carpinelli Table 5.2, as it appears in the text: SHIFT MICRO-OPERATIONS

Original figure or table © 2001
by Addison Wesley Longman, Inc

Carpinelli Table 5.2, *CORRECTED*: SHIFT MICRO-OPERATIONS

The corrected version corresponds to the definitions of the various types of shift operations typically implemented in the Arithmetic Logic Unit (ALU) portion of Central Processing Units (CPUs).

Operation	Notation
Logical ~~Linear~~ shift left	shl(X)
Logical ~~Linear~~ shift right	shr(X)
Circular shift left	cil(X)
Circular shift right	cir(X)
Arithmetic shift left	ashl(X)
Arithmetic shift right	ashr(X)
Decimal shift left	dshl(X)
Decimal shift right	dshr(X)

Original figure or table © 2001 by Addison Wesley Longman, Inc

Carpinelli Figure 5.7:
D FLIP-FLOP

Original figure or table © 2001
by Addison Wesley Longman, Inc

© 2003 Charles Abzug

Carpinelli Figure 5.8:
DATA PATHS to IMPLEMENT *RTL* CODE USING DIRECT CONNECTIONS

Original figure or table © 2001
by Addison Wesley Longman, Inc

Carpinelli Figure 5.9: COMPLETE DESIGN IMPLEMENTING *RTL* CODE USING DIRECT CONNECTIONS

Original figure or table © 2001 by Addison Wesley Longman, Inc

Carpinelli Figure 5.10: IMPLEMENTATION of *RTL* CODE USING a BUS and TRI-STATE BUFFERS

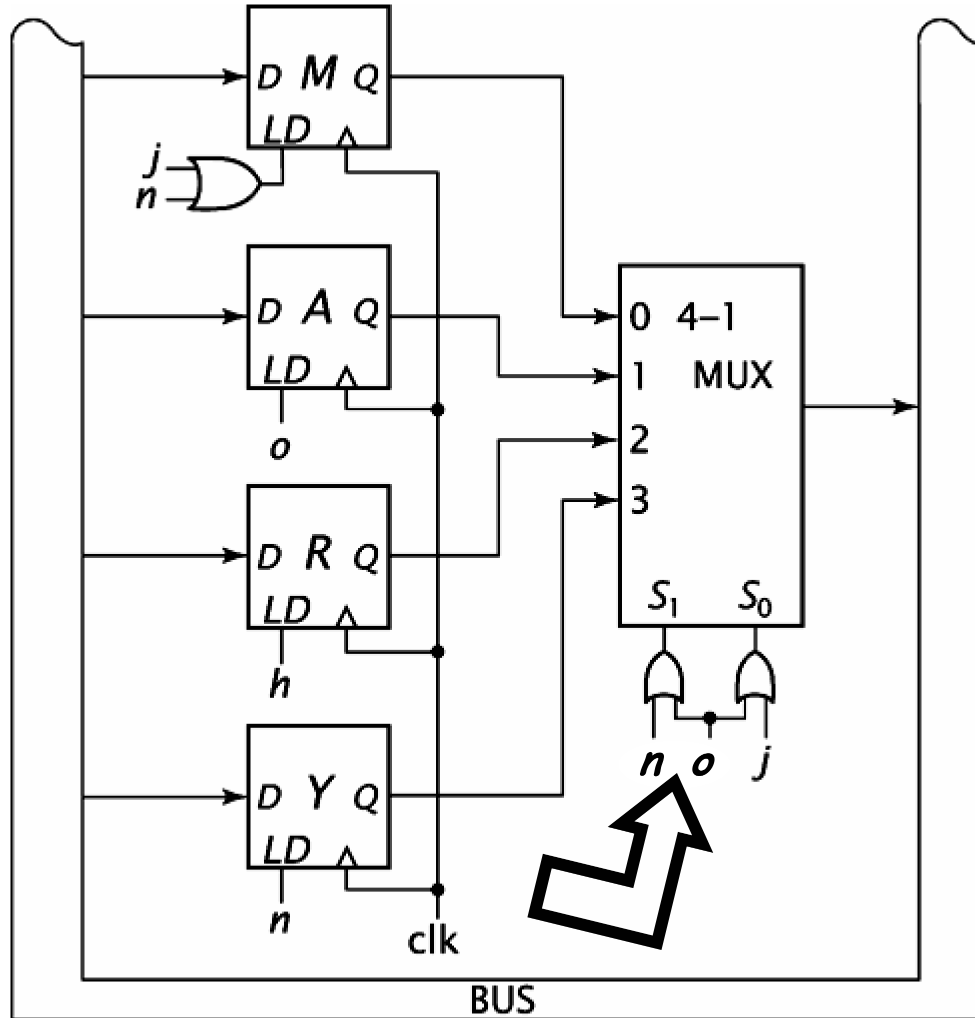
**Original figure or table © 2001
by Addison Wesley Longman, Inc**

Carpinelli Figure 5.11, as it appears in the text: IMPLEMENTATION of *RTL* CODE USING a BUS and a MULTIPLEXER

Original figure or table © 2001 by Addison Wesley Longman, Inc

Carpinelli Figure 5.11, *CORRECTED*: IMPLEMENTATION of *RTL* CODE USING a BUS and a MULTIPLEXER

Original figure or table © 2001 by Addison Wesley Longman, Inc



CPU Design

© 2003 Charles Abzug

Major Design Principle

1. Define what the CPU must do.

2. Match its capabilities to its job:
   a) Instruction Set Architecture:
      instructions,
      addressing modes,
      programmer-accessible register set
   b) Internal Registers (not programmer-accessible)
   c) State Diagram, Micro-Operations, internal Data Paths, and Control
   d) Control Logic to implement

© 2003 Charles Abzug

Carpinelli Figure 6.1:
GENERIC STATE DIAGRAM for a CPU

Original figure or table © 2001
by Addison Wesley Longman, Inc

Carpinelli Table 6.1:
INSTRUCTION SET for Carpinelli's "Very Simple" CPU

Original figure or table © 2001
by Addison Wesley Longman, Inc

Carpinelli Figure 6.2: FETCH CYCLE for Carpinelli's "VERY SIMPLE" CPU

AR←PC

DR←M, PC←PC+1

IR←DR[7..6], AR←DR[5..0]

Original figure or table © 2001
by Addison Wesley Longman, Inc

Carpinelli Figure 6.3: FETCH and DECODE CYCLES for Carpinelli's "Very Simple" CPU

Original figure or table © 2001
by Addison Wesley Longman, Inc

Carpinelli Figure 6.4: COMPLETE STATE DIAGRAM for Carpinelli's "Very Simple" CPU

ADD1: DR←M

ADD2: AC←AC+DR

AND1: DR←M

AND2: AC←AC∧DR

JMP1: PC←DR[5..0]

INC1: AC←AC+1

Original figure or table © 2001
by Addison Wesley Longman, Inc

Carpinelli Figure 6.5: PRELIMINARY REGISTER SECTION for Carpinelli's "Very Simple" CPU

ADD1: DR←M
AND1: DR←M
FETCH2: *DR←M*

Original figure or table © 2001
by Addison Wesley Longman, Inc

FETCH2: *PC←PC+1*
JMP1: PC←DR[5..0]

FETCH1: *AR←PC*
FETCH3: *AR←DR[5..0]*

FETCH3: *IR←DR[7..6]*

ADD2: AC←AC+DR
AND2: AC←AC∧DR
INC1: AC←AC+1

Carpinelli Figure 6.6: FINAL REGISTER SECTION for Carpinelli's "Very Simple" CPU

Original figure or table © 2001
by Addison Wesley Longman, Inc

ADD1: DR←M
AND1: DR←M
FETCH2: *DR←M*

FETCH2: *PC←PC+1*
JMP1: PC←DR[5..0]

FETCH1: *AR←PC*
FETCH3: *AR←DR[5..0]*

FETCH3: *IR←DR[7..6]*

ADD2: AC←AC+DR
AND2: AC←AC∧DR
INC1: AC←AC+1

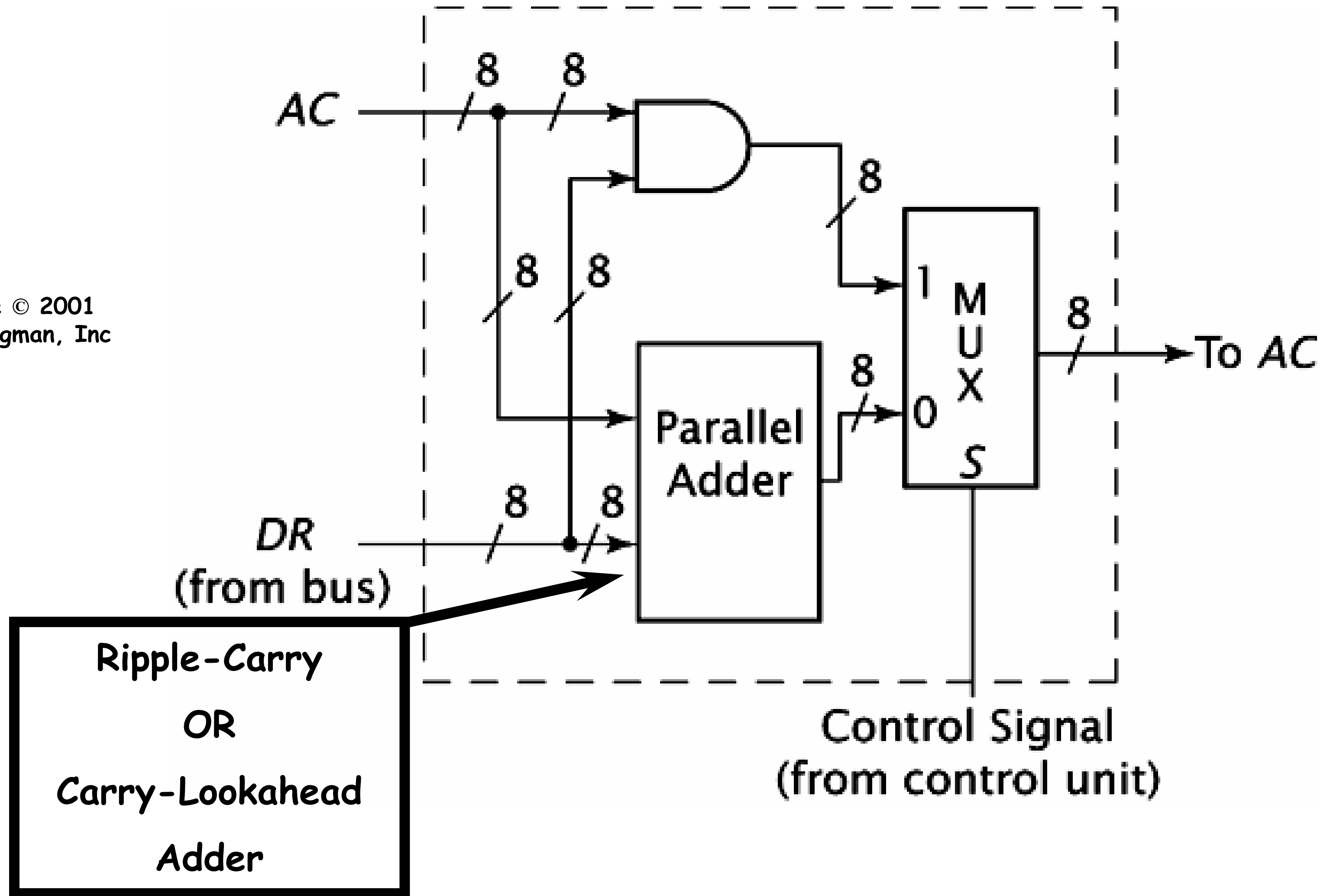
© 2003 Charles Abzug

Carpinelli Figure 6.7:
A “Very Simple” ALU

**Original figure or table © 2001
by Addison Wesley Longman, Inc**

Carpinelli Figure 6.7: A "Very Simple" ALU

Original figure or table © 2001
by Addison Wesley Longman, Inc



© 2003 Charles Abzug

Carpinelli Figure 6.8:
GENERIC HARDWIRED CONTROL UNIT

Original figure or table © 2001
by Addison Wesley Longman, Inc

Carpinelli Table 6.2: INSTRUCTIONS, FIRST STATES, and OPCODES for Carpinelli's "Very Simple" CPU

Original figure or table © 2001
by Addison Wesley Longman, Inc

Carpinelli Table 6.3: COUNTER VALUES for the INITIALLY PROPOSED MAPPING FUNCTION

Original figure or table © 2001
by Addison Wesley Longman, Inc

Carpinelli Table 6.3, *AUGMENTED*: COUNTER VALUES for the FINAL MAPPING FUNCTION

IR[1..0]	Counter Value	State
00	1000	ADD1
01	1010	AND1
10	1100	JMP1
11	1110	INC1

Original figure or table © 2001
by Addison Wesley Longman, Inc

Carpinelli Figure 6.9: HARDWIRED CONTROL UNIT for Carpinelli's "Very Simple" CPU

Original figure or table © 2001
by Addison Wesley Longman, Inc

Carpinelli Figure 6.10: GENERATION of CONTROL SIGNALS for Carpinelli's "Very Simple" CPU

Original figure or table © 2001
by Addison Wesley Longman, Inc

Carpinelli Table 6.4: MICROINSTRUCTION EXECUTION TRACE for ALL INSTRUCTIONS of Carpinelli's "VERY SIMPLE" CPU

Original figure or table © 2001
by Addison Wesley Longman, Inc

Carpinelli Figure 6.11: FETCH and DECODE CYCLES for Carpinelli's "Relatively Simple" CPU

Original figure or table © 2001
by Addison Wesley Longman, Inc

Carpinelli Table 6.5: INSTRUCTION SET for Carpinelli's "Relatively Simple" CPU

Original figure or table © 2001
by Addison Wesley Longman, Inc

MICROINSTRUCTIONS for Carpinelli's "RELATIVELY SIMPLE CPU"

FETCH1: *AR←PC*
FETCH2: *DR←M, PC←PC+1*
FETCH3: *IR←DR, AR←PC*

NOP1: —

ADD1: *AC←AC+R, If (AC+R = 0) then Z←1, else Z←0*

SUB1: *AC←AC-R, If (AC-R = 0) then Z←1, else Z←0*

INAC1: *AC←AC+1, If (AC+1 = 0) then Z←1, else Z←0*

AND1: *AC←AC∧R, If (AC∧R = 0) then Z←1, else Z←0*

OR1: *AC←AC∨R, If (AC∨R = 0) then Z←1, else Z←0*

XOR1: *AC←AC⊕R, If (AC⊕R = 0) then Z←1, else Z←0*

NOT1: *AC←!AC, If (!AC = 0) then Z←1, else Z←0*

MVAC1: *R←AC*

MVR1: *AC←R*

CLAC1: *AC←0, Z←1*

© 2003 Charles Abzug

MICROINSTRUCTIONS for Carpinelli's "RELATIVELY SIMPLE CPU" (continued)

LDAC1: *DR←M, PC←PC+1, AR←AR+1*
LDAC2: *TR←DR, DR←M, PC←PC+1*
LDAC3: *AR←DR, TR*
LDAC4: *DR←M*
LDAC5: *AC←DR*

STAC1: *DR←M, PC←PC+1, AR←AR+1*
STAC2: *TR←DR, DR←M, PC←PC+1*
STAC3: *AR←DR, TR*
STAC4: *DR←AC*
STAC5: *M←DR*

JUMP1: *DR←M, AR←AR+1*
JUMP2: *TR←DR, DR←M*
JUMP3: *PC←DR, TR*

JMPZY1: *DR←M, AR←AR+1*
JMPZY2: *TR←DR, DR←M*
JMPZY3: *PC←DR, TR*

JPNZY1: *DR←M, AR←AR+1*
JPNZY2: *TR←DR, DR←M*
JPNZY3: *PC←DR, TR*

JMPZN1: *PC←PC+1*
JMPZN2: *PC←PC+1*

JPNZN1: *PC←PC+1*
JPNZN2: *PC←PC+1*

© 2003 Charles Abzug

Carpinelli Figure 6.12: COMPLETE STATE DIAGRAM for Carpinelli's "Relatively Simple" CPU

Original figure or table © 2001
by Addison Wesley Longman, Inc

Carpinelli Figure 6.13: PRELIMINARY REGISTER SECTION for Carpinelli's "Relatively Simple" CPU

Original figure or table © 2001
by Addison Wesley Longman, Inc

Carpinelli Figure 6.14: BIDIRECTIONAL DATA PIN (GENERIC)

**Original figure or table © 2001
by Addison Wesley Longman, Inc**

Carpinelli Figure 6.15: FINAL REGISTER SECTION for Carpinelli's "Relatively Simple" CPU

Original figure or table © 2001
by Addison Wesley Longman, Inc

Carpinelli Figure 6.16: ARITHMETIC LOGIC UNIT for Carpinelli's "Relatively Simple" CPU

Original figure or table © 2001 by Addison Wesley Longman, Inc

Carpinelli Figure 6.17: HARDWIRED CONTROL UNIT for Carpinelli's "Relatively Simple" CPU

Original figure or table © 2001
by Addison Wesley Longman, Inc

Carpinelli Table 6.6: STATE DEFINITIONS for Carpinelli's "RELATIVELY SIMPLE CPU"

Original figure or table © 2001
by Addison Wesley Longman, Inc

Carpinelli Table 6.7: CONTROL SIGNAL VALUES for Carpinelli's "RELATIVELY SIMPLE CPU"

Original figure or table © 2001
by Addison Wesley Longman, Inc

Carpinelli Figure 6.18 (a): REGISTER SECTION for Carpinelli's "Relatively Simple" CPU Using MULTIPLE BUSES

Original figure or table © 2001
by Addison Wesley Longman, Inc

Carpinelli Figure 6.18 (b): REGISTER SECTION for Carpinelli's "Relatively Simple" CPU Using MULTIPLE BUSES

Original figure or table © 2001
by Addison Wesley Longman, Inc

Carpinelli Figure 6.19: INTERNAL ORGANIZATION of the Intel 8085 Processor

Original figure or table © 2001
by Addison Wesley Longman, Inc

END

© 2003 Charles Abzug