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


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.
STATEMENTS in ASSEMBLY LANGUAGE: Three Types

1. Machine Instructions:

2. Assembler Directives:

3. Macro Expansion Commands:
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.
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.
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.
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].
INSTRUCTION-EXECUTION CYCLE - Version 2

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. Is there an INTERRUPT? If so, then service it.

5. GOTO Fetch.
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
DATA-COPY (DATA-TRANSFER) INSTRUCTIONS in Carpinelli’s “Relatively Simple CPU”

<table>
<thead>
<tr>
<th>Machine Code</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0001</td>
<td>LDAC</td>
<td>Load (i.e., copy into) the Accumulator with the contents of either a Main Memory location or data from an input device.</td>
</tr>
<tr>
<td>0000 0010</td>
<td>STAC</td>
<td>Store (i.e., copy the contents of) the Accumulator to Main Memory or to an output device.</td>
</tr>
<tr>
<td>0000 0011</td>
<td>MVAC</td>
<td>Move (i.e., copy) the contents of the Accumulator to Register R.</td>
</tr>
<tr>
<td>0000 0100</td>
<td>MOVR</td>
<td>Move (i.e., copy) the contents of Register R to the Accumulator.</td>
</tr>
<tr>
<td>0000 0000</td>
<td>NOP</td>
<td>No Operation</td>
</tr>
</tbody>
</table>
# DATA-OPERATION INSTRUCTIONS in Carpinelli’s “Relatively Simple CPU”

<table>
<thead>
<tr>
<th>Machine Code</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 1000</td>
<td>ADD</td>
<td>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.</td>
</tr>
<tr>
<td>0000 1001</td>
<td>SUB</td>
<td>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.</td>
</tr>
<tr>
<td>0000 1010</td>
<td>INAC</td>
<td>Increment the contents of the Accumulator, and adjust the value of the $Z$ bit.</td>
</tr>
<tr>
<td>0000 1011</td>
<td>CLAC</td>
<td>Clear the contents of the Accumulator, and set the $Z$ bit.</td>
</tr>
<tr>
<td>0000 1100</td>
<td>AND</td>
<td>Bitwise “AND” the contents of the Accumulator with the contents of Register $R$, and adjust the value of the $Z$ bit.</td>
</tr>
<tr>
<td>0000 1101</td>
<td>OR</td>
<td>Bitwise “OR” the contents of the Accumulator with the contents of Register $R$, and adjust the value of the $Z$ bit.</td>
</tr>
<tr>
<td>0000 1110</td>
<td>XOR</td>
<td>Bitwise “XOR” the contents of the Accumulator with the contents of Register $R$, and adjust the value of the $Z$ bit.</td>
</tr>
<tr>
<td>0000 1111</td>
<td>NOT</td>
<td>Complement the contents of the Accumulator, and adjust the value of the $Z$ bit.</td>
</tr>
</tbody>
</table>
## PROGRAM-CONTROL INSTRUCTIONS
### in Carpinelli’s “Relatively Simple CPU”

<table>
<thead>
<tr>
<th>Machine Code</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0101</td>
<td>JUMP</td>
<td>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.</td>
</tr>
</tbody>
</table>
| 0000 0110    | 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    | 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
MODES of ADDRESSING

1. Direct Addressing Mode: Memory address is explicitly stated within the instruction, after the op code.
   EXAMPLE: \textit{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 not the address of the operand, but rather is the address of the address of the operand.
   EXAMPLE: \textit{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 value of the operand is located within the specified register.
   EXAMPLE: \textit{LDAC register-identifier, e.g.: LDAC R}
   EXECUTION: Copy into the Accumulator the contents of Register R.

4. Register Indirect Addressing Mode: The memory address of the operand is located within the specified register.
   EXAMPLE: \textit{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.
MODES of ADDRESSING (continued)

5. Immediate Addressing Mode: The actual value of the operand is stated within the instruction.
   EXAMPLE: \textit{LDAC \#actual-value}, e.g.: \textit{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: \textit{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 offset of the desired memory location from the current contents of the Program Counter (NOTE: This is \textit{NOT} the offset from the location of the currently-executing instruction, but rather the offset from the location of the \textit{NEXT} instruction in sequence following the currently-executing instruction).
   EXAMPLE: \textit{JMP \$offset-amount}, e.g.: \textit{JMP \#3C}
   EXECUTION: If the \textit{JMP} instruction starts at memory address F000, then the next instruction located after the \textit{JMP} will be located at F002, and after execution of the \textit{JMP}, the next instruction to be executed is at address F03E (= F002 + 3C)
6. Indexed Addressing Mode: The numeric value specified within the instruction gives the base address 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)
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:

- a) 0: LDAC 5
  - Instruction gets data from location 5
  - 5: 10 → stores value in CPU

Indirect addressing mode:

- b) 0: LDAC @ 5
  - Instruction gets address from location 5
  - 5: 10
  - then gets data from location 10
  - 10: 20 → stores value in CPU

Register Direct addressing mode:

- c) 0: LDAC R
  - Instruction gets data from register R
  - R: 5 → stores value in CPU

Register Indirect addressing mode:

- d) 0: LDAC R @ R or LDAC (R)
  - Instruction gets address from register R
  - R: 5
  - then gets data from location 5
  - 5: 10 → stores value in CPU

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:
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

g) 0: LDAC $5

Relative addressing mode:
1: ➔ instruction adds address of next instruction (1) to
   ➔ offset (5) to get address (6)
5: 12 ➔ stores value in CPU

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
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 next instruction (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
### Carpinelli Figure 3.4:
**INSTRUCTION CODE FORMATS, ASSEMBLY LANGUAGE, and MACHINE CODE**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD A,B,C (A=B+C)</td>
<td>1010 00 01 10</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVE A,B (A=B)</td>
<td>1000 00 01</td>
<td></td>
</tr>
<tr>
<td>ADD A,C (A=A+C)</td>
<td>1010 0 10</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD B (Acc=B)</td>
<td>0000 01</td>
<td></td>
</tr>
<tr>
<td>ADD C (Acc=Acc+C)</td>
<td>1010 0</td>
<td></td>
</tr>
<tr>
<td>STORE A (A=Acc)</td>
<td>0001 00</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH B (Stack=B)</td>
<td>0101</td>
<td></td>
</tr>
<tr>
<td>PUSH C (Stack=C,B)</td>
<td>0110</td>
<td></td>
</tr>
<tr>
<td>ADD (Stack=B+C)</td>
<td>1010</td>
<td></td>
</tr>
<tr>
<td>POP A (A=stack)</td>
<td>1100</td>
<td></td>
</tr>
</tbody>
</table>
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.


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.
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
Carpinelli Table 3.1:
INSTRUCTION SET
for the “RELATIVELY SIMPLE CPU”
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
IMPLEMENTATION of a HIGH-LEVEL-LANGUAGE STATEMENT in ASSEMBLY LANGUAGE:
\[ X = A + (B \times C) + D \]

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\times 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\times 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.
IMPLEMENTATION of a HIGH-LEVEL-LANGUAGE STATEMENT in ASSEMBLY LANGUAGE:

\[ X = A + (B \times C) + D \]

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.
IMPLEMENTATION of a HIGH-LEVEL-LANGUAGE STATEMENT in ASSEMBLY LANGUAGE:
\[ X = A + (B \times C) + D \]

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLAC</td>
<td>% Create in the Accumulator a content of 0.</td>
</tr>
<tr>
<td>STAC X</td>
<td>% Initialize the value of final result to 0.</td>
</tr>
<tr>
<td>STAC Count</td>
<td>% Initialize the value of the counter to 0.</td>
</tr>
<tr>
<td>Loop:</td>
<td>% (continued)</td>
</tr>
<tr>
<td>LDAC B</td>
<td>% Copy the value of the multiplicand to the Accumulator.</td>
</tr>
<tr>
<td>MVAC</td>
<td>% Copy the multiplicand to Register R.</td>
</tr>
<tr>
<td>LDAC X</td>
<td>% Load current value of end-result into the Accumulator.</td>
</tr>
<tr>
<td>ADD</td>
<td>% Add value of multiplicand to end-result.</td>
</tr>
<tr>
<td>STAC X</td>
<td>% Copy out the current value of end-result.</td>
</tr>
<tr>
<td>LDAC Count</td>
<td>% Prepare to update count.</td>
</tr>
<tr>
<td>INAC</td>
<td>% Update the count.</td>
</tr>
<tr>
<td>STAC Count</td>
<td>% Store the updated count.</td>
</tr>
<tr>
<td>MVAC</td>
<td>% Copy the updated count to Register R.</td>
</tr>
<tr>
<td>LDAC C</td>
<td>% Load the value of the multiplier.</td>
</tr>
<tr>
<td>SUB</td>
<td>% Compare the current count to the multiplier.</td>
</tr>
<tr>
<td>JNZ Loop</td>
<td>% Continue multiplying.</td>
</tr>
</tbody>
</table>
IMPLEMENTATION of a HIGH-LEVEL-LANGUAGE STATEMENT in ASSEMBLY LANGUAGE:

\[ X = A + (B \times C) + D \]

% Continuation of ONE-Operand ASL code:

% Finished multiplying; X contains B\times 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\times C) into the Accumulator.
ADD % Add to B\times C the value of A.
STAC X % Copy back to memory the current value of end-result (= A + B\times 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\times C) into the Accumulator.
ADD % Add to A + B\times C the value of D.
STAC X % Copy out the final answer (= A + B\times C + D).
IMPLEMENTATION of a HIGH-LEVEL-LANGUAGE STATEMENT in ASSEMBLY LANGUAGE:

\[ X = A + (B \times C) + D \]

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

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV</td>
<td>Move data</td>
</tr>
<tr>
<td>ADD</td>
<td>Add data</td>
</tr>
<tr>
<td>SUB</td>
<td>Subtract</td>
</tr>
<tr>
<td>AND</td>
<td>AND data</td>
</tr>
<tr>
<td>OR</td>
<td>OR data</td>
</tr>
<tr>
<td>XOR</td>
<td>XOR data</td>
</tr>
</tbody>
</table>

Original figure or table © 2001 by Addison Wesley Longman, Inc
Carpinelli Table 3.5:
PROGRAM CONTROL INSTRUCTIONS
for the Intel 8085 MICROPROCESSOR
Carpinelli Table 3.6:
EXECUTION TRACE
of the LOOP SUMMATION PROGRAM
for the Intel 8085 MICROPROCESSOR
Overview of Computer Organization
Carpinelli Figure 4.1:
GENERIC COMPUTER ORGANIZATION
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*. 
MAJOR COMPONENTS of a BUS

1. Address Lines
2. Data Lines
3. Control Lines
4. Power Lines
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
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.
Carpinelli Figure 4.2:
TIMING DIAGRAMS for MEMORY BUS OPERATIONS
Carpinelli Figure 4.3:
INTERNAL ORGANIZATION of the CPU
Carpinelli Figure 4.4, as it appears in the text:
INTERNAL ORGANIZATION of a LINEAR 8x2 ROM CHIP
Carpinelli Figure 4.4,
CORRECTED:
INTERNAL ORGANIZATION of a LINEAR 8x2 ROM CHIP
Carpinelli Figure 4.5:
INTERNAL TWO-DIMENSIONAL ORGANIZATION of an 8x2 ROM CHIP
Carpinelli Figure 4.6:  
CONSTRUCTION of an 8x4 MEMORY SUBSYSTEM from TWO 8x2 ROM CHIPS
Carpinelli Figure 4.8:
CONSTRUCTION of an 8x4 MEMORY SUBSYSTEM
from TWO 8x2 ROM CHIPS, CONTROL SIGNALS ADDED
Carpinelli Figure 4.7 (a):
CONSTRUCTION of a 16x2 MEMORY SUBSYSTEM
from TWO 8x2 ROM CHIPS, SEQUENTIAL ADDRESSING
Carpinelli Figure 4.7 (b):
CONSTRUCTION of a 16x2 MEMORY SUBSYSTEM
from TWO 8x2 ROM CHIPS, INTERLEAVED ADDRESSING
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.
Carpinelli Table 4.1:
ALTERNATIVE CONVENTIONS for REPRESENTATION
of MULTIPLE-BYTE DATA

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

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Data (in hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>01</td>
</tr>
<tr>
<td>101</td>
<td>02</td>
</tr>
<tr>
<td>102</td>
<td>03</td>
</tr>
<tr>
<td>103</td>
<td>04</td>
</tr>
</tbody>
</table>

Big-Endian

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Data (in hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>04</td>
</tr>
<tr>
<td>101</td>
<td>03</td>
</tr>
<tr>
<td>102</td>
<td>02</td>
</tr>
<tr>
<td>103</td>
<td>01</td>
</tr>
</tbody>
</table>

Little-Endian
Carpinelli Figure 4.9: An INPUT DEVICE
Carpinelli Figure 4.10: An OUTPUT DEVICE
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”
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”
Carpinelli Figure 4.15:
DEMULTIPLEXING of the AD PINS
for the Intel 8085 MICROPROCESSOR
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 I/O 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-STATE. THESE RESISTORS ARE NOT INCLUDED ON THE PC BOARD LAYOUT OF FIGURE 3-7.
Carpinelli Figure 4.16:
A Minimal 8085-based Computer System:
MAGNIFIED PARTIAL DETAIL
Carpinelli Figure 4.16: A Minimal 8085-based Computer System: MAGNIFIED PARTIAL DETAIL
Register Transfer Language (RTL)

and

Micro-Operations (Micro-Ops)
Carpinelli Figure 5.1, as it appears in the text: IMPLEMENTATION of the MICRO-OPERATION \( X \leftarrow Y \)
Carpinelli Figure 5.1, CORRECTED:
IMPLEMENTATION of the MICRO-OPERATION $X \leftarrow Y$

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 \[ \alpha: X \leftarrow Y \]
Carpinelli Figure 5.2, CORRECTED: IMPLEMENTATION of the DATA TRANSFER

\[ \alpha : X \leftarrow Y \]

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.
Carpinelli Figure 5.3, CORRECTED: 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 5.4:
IMPLEMENTATION of the DATA TRANSFER
\[ \alpha : X \leftarrow Y, \ Z \leftarrow Y \]
Carpinelli Figure 5.5:
IMPLEMENTATION of the DATA TRANSFER
\[ \alpha : X \leftarrow 0, \beta : X \leftarrow 1 \]
Carpinelli Figure 5.6: IMPLEMENTATION of the FOUR-BIT DATA TRANSFER
\[ \alpha: X \leftarrow Y \]
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).

<table>
<thead>
<tr>
<th>Operation</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical shift left</td>
<td>shl(X)</td>
</tr>
<tr>
<td>Logical shift right</td>
<td>shr(X)</td>
</tr>
<tr>
<td>Circular shift left</td>
<td>cil(X)</td>
</tr>
<tr>
<td>Circular shift right</td>
<td>cir(X)</td>
</tr>
<tr>
<td>Arithmetic shift left</td>
<td>ashl(X)</td>
</tr>
<tr>
<td>Arithmetic shift right</td>
<td>ashr(X)</td>
</tr>
<tr>
<td>Decimal shift left</td>
<td>dshl(X)</td>
</tr>
<tr>
<td>Decimal shift right</td>
<td>dshr(X)</td>
</tr>
</tbody>
</table>

Original figure or table © 2001 by Addison Wesley Longman, Inc
Carpinelli Figure 5.7: 
D FLIP-FLOP
Carpinelli Figure 5.8:
DATA PATHS to IMPLEMENT RTL CODE
USING DIRECT CONNECTIONS
Cerinelli 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
Carpinelli Figure 5.11, CORRECTED: IMPLEMENTATION of RTL CODE USING a BUS and a MULTIPLEXER
CPU Design
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
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
Carpinelli Figure 6.2: 
FETCH CYCLE for Carpinelli’s “VERY SIMPLE” CPU

\[ AR \leftarrow PC \]

\[ DR \leftarrow M, \ PC \leftarrow PC + 1 \]

\[ IR \leftarrow DR[7..6], \ AR \leftarrow DR[5..0] \]
Carpinelli Figure 6.3: FETCH and DECODE CYCLES for Carpinelli’s “Very Simple” CPU
Carpinelli Figure 6.4: COMPLETE STATE DIAGRAM for Carpinelli’s “Very Simple” CPU

ADD1: \( DR \leftarrow M \)
ADD2: \( AC \leftarrow AC + DR \)

AND1: \( DR \leftarrow M \)
AND2: \( AC \leftarrow AC \land DR \)

JMP1: \( PC \leftarrow DR[5..0] \)

INC1: \( AC \leftarrow AC + 1 \)
Carpinelli Figure 6.5:
PRELIMINARY REGISTER
SECTION
for Carpinelli’s
“Very Simple” CPU

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
Carpinelli Figure 6.6: FINAL REGISTER SECTION for Carpinelli’s “Very Simple” CPU

ADD1: \( DR \leftarrow M \)
AND1: \( DR \leftarrow M \)
FETCH2: \( DR \leftarrow M \)

FETCH2: \( PC \leftarrow PC+1 \)
JMP1: \( PC \leftarrow DR[5..0] \)

FETCH1: \( AR \leftarrow PC \)
FETCH3: \( AR \leftarrow DR[5..0] \)

FETCH3: \( IR \leftarrow DR[7..6] \)

ADD2: \( AC \leftarrow AC+DR \)
AND2: \( AC \leftarrow AC \land DR \)
INC1: \( AC \leftarrow AC+1 \)
Carpinelli Figure 6.7: 
A “Very Simple” ALU
Carpinelli Figure 6.7:
A “Very Simple” ALU

Original figure or table © 2001 by Addison Wesley Longman, Inc
Carpinelli Figure 6.8:
GENERIC HARDWIRED CONTROL UNIT
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

<table>
<thead>
<tr>
<th>$IR[1..0]$</th>
<th>Counter Value</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1000</td>
<td>ADD1</td>
</tr>
<tr>
<td>01</td>
<td>1010</td>
<td>AND1</td>
</tr>
<tr>
<td>10</td>
<td>1100</td>
<td>JMP1</td>
</tr>
<tr>
<td>11</td>
<td>1110</td>
<td>INC1</td>
</tr>
</tbody>
</table>
Carpinelli Figure 6.9: HARDWIRED CONTROL UNIT for Carpinelli’s “Very Simple” CPU
Carpinelli Figure 6.10: GENERATION of CONTROL SIGNALS for Carpinelli’s “Very Simple” CPU
Carpinelli Table 6.4:
MICROINSTRUCTION EXECUTION TRACE
for ALL INSTRUCTIONS of Carpinelli’s “VERY SIMPLE” CPU
Carpinelli Figure 6.11: FETCH and DECODE CYCLES for Carpinelli’s “Relatively Simple” CPU
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: \( AR \leftarrow PC \)
FETCH2: \( DR \leftarrow M, PC \leftarrow PC + 1 \)
FETCH3: \( IR \leftarrow DR, AR \leftarrow PC \)

NOP1: —

ADD1: \( AC \leftarrow AC + R, \text{ If } (AC + R = 0) \text{ then } Z \leftarrow 1, \text{ else } Z \leftarrow 0 \)

SUB1: \( AC \leftarrow AC - R, \text{ If } (AC - R = 0) \text{ then } Z \leftarrow 1, \text{ else } Z \leftarrow 0 \)

INAC1: \( AC \leftarrow AC + 1, \text{ If } (AC + 1 = 0) \text{ then } Z \leftarrow 1, \text{ else } Z \leftarrow 0 \)

AND1: \( AC \leftarrow AC \land R, \text{ If } (AC \land R = 0) \text{ then } Z \leftarrow 1, \text{ else } Z \leftarrow 0 \)

OR1: \( AC \leftarrow AC \lor R, \text{ If } (AC \lor R = 0) \text{ then } Z \leftarrow 1, \text{ else } Z \leftarrow 0 \)

XOR1: \( AC \leftarrow AC \oplus R, \text{ If } (AC \oplus R = 0) \text{ then } Z \leftarrow 1, \text{ else } Z \leftarrow 0 \)

NOT1: \( AC \leftarrow \lnot AC, \text{ If } (\lnot AC = 0) \text{ then } Z \leftarrow 1, \text{ else } Z \leftarrow 0 \)

MVAC1: \( R \leftarrow AC \)
MVR1: \( AC \leftarrow R \)
CLAC1: \( AC \leftarrow 0, Z \leftarrow 1 \)
MICROINSTRUCTIONS for Carpinelli’s “RELATIVELY SIMPLE CPU” (continued)

LDAC1: \(DR \leftarrow M, PC \leftarrow PC+1, AR \leftarrow AR+1\)  
LDAC2: \(TR \leftarrow DR, DR \leftarrow M, PC \leftarrow PC+1\)  
LDAC3: \(AR \leftarrow DR, TR\)  
LDAC4: \(DR \leftarrow M\)  
LDAC5: \(AC \leftarrow DR\)  
STAC1: \(DR \leftarrow M, PC \leftarrow PC+1, AR \leftarrow AR+1\)  
STAC2: \(TR \leftarrow DR, DR \leftarrow M, PC \leftarrow PC+1\)  
STAC3: \(AR \leftarrow DR, TR\)  
STAC4: \(DR \leftarrow AC\)  
STAC5: \(M \leftarrow DR\)  
JUMP1: \(DR \leftarrow M, AR \leftarrow AR+1\)  
JUMP2: \(TR \leftarrow DR, DR \leftarrow M\)  
JUMP3: \(PC \leftarrow DR, TR\)  
JMPZY1: \(DR \leftarrow M, AR \leftarrow AR+1\)  
JMPZY2: \(TR \leftarrow DR, DR \leftarrow M\)  
JMPZY3: \(PC \leftarrow DR, TR\)  
JMPZN1: \(PC \leftarrow PC+1\)  
JMPZN2: \(PC \leftarrow PC+1\)  
JPNZY1: \(DR \leftarrow M, AR \leftarrow AR+1\)  
JPNZY2: \(TR \leftarrow DR, DR \leftarrow M\)  
JPNZY3: \(PC \leftarrow DR, TR\)  
JPNZN1: \(PC \leftarrow PC+1\)  
JPNZN2: \(PC \leftarrow PC+1\)
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
Carpinelli Figure 6.14: BIDIRECTIONAL DATA PIN (GENERIC)
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
Carpinelli Figure 6.17: HARDWIRED CONTROL UNIT for Carpinelli’s “Relatively Simple” CPU
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
Carpinelli Figure 6.18 (b): REGISTER SECTION for Carpinelli’s “Relatively Simple” CPU Using MULTIPLE BUSES
Carpinelli Figure 6.19: INTERNAL ORGANIZATION of the Intel 8085 Processor

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