Instruction Codes

- An instruction code is a group of bits that instruct the computer to perform a specific operation.
- The operation code of an instruction is a group of bits that define operations such as addition, subtraction, shift, complement, etc.
- An instruction must also include one or more operands, which indicate the registers and/or memory addresses from which data is taken or to which data is deposited.
Microoperations

• The instructions are stored in computer memory in the same manner that data is stored.
• The control unit interprets these instructions and uses the operations code to determine the sequences of microoperations that must be performed to execute the instruction.

Stored Program Organization

• The operands are specified by indicating the registers and/or memory locations in which they are stored.
  – $k$ bits can be used to specify which of $2^k$ registers (or memory locations) are to be used.
• The simplest design is to have one processor register (called the accumulator) and two fields in the instruction, one for the opcode and one for the operand.
• Any operation that does not need a memory operand frees the other bits to be used for other purposes, such as specifying different operations.
Stored Program Organization

<table>
<thead>
<tr>
<th>15 12 11</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>Address</td>
</tr>
</tbody>
</table>

Instruction format

<table>
<thead>
<tr>
<th>15 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary operand</td>
</tr>
</tbody>
</table>

Memory
4096 x 16

Instructions (programs)

Operands (data)

Processor Register (accumulator or AC)

Addressing Modes

- There are four different types of operands that can appear in an instruction:
  - **Direct operand** - an operand stored in the register or in the memory location specified.
  - **Indirect operand** - an operand whose address is stored in the register or in the memory location specified.
  - **Immediate operand** - an operand whose **value** is specified in the instruction.
Direct and Indirect Addressing

Instruction format

Direct addressing

Indirect addressing

Registers

- Computer instructions are stored in consecutive locations and are executed sequentially; this requires a register which can store the address of the next instruction; we call it the Program Counter.
- We need registers which can hold the address at which a memory operand is stored as well as the value itself.
- We need a place where we can store
  - temporary data
  - the instruction being executed,
  - a character being read in
  - a character being written out.
## List of Registers for the Basic Computer

<table>
<thead>
<tr>
<th>Register Symbol</th>
<th># of Bits</th>
<th>Register Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>DR</td>
<td>16</td>
<td>Data Register</td>
<td>Holds memory operand</td>
</tr>
<tr>
<td>AR</td>
<td>12</td>
<td>Address Register</td>
<td>Holds mem. address</td>
</tr>
<tr>
<td>AC</td>
<td>16</td>
<td>Accumulator</td>
<td>Processor Reg.</td>
</tr>
<tr>
<td>IR</td>
<td>16</td>
<td>Instruction Register</td>
<td>Holds instruction code</td>
</tr>
<tr>
<td>PC</td>
<td>12</td>
<td>Program Counter</td>
<td>Holds instruction address</td>
</tr>
<tr>
<td>TR</td>
<td>16</td>
<td>Temporary Register</td>
<td>Holds temporary data</td>
</tr>
<tr>
<td>INPR</td>
<td>8</td>
<td>Input Register</td>
<td>Holds input character</td>
</tr>
<tr>
<td>OUTR</td>
<td>8</td>
<td>Output Register</td>
<td>Holds output character</td>
</tr>
</tbody>
</table>

## Basic Computer Registers and Memory

- **PC**: 011
- **AR**: 011
- **IR**: 015
- **TR**: 015
- **OUTR**: 0 7
- **INPR**: 0 7
- **DR**: 015
- **AC**: 015

**Memory**
- 4096 words
- 16 bits per word
The Common Bus

- To avoid excessive wiring, memory and all the register are connected via a common bus.
- The specific output that is selected for the bus is determined by $S_2S_1S_0$.
- The register whose LD (Load) is enable receives the data from the bus.
- Registers can be incremented by setting the INR control input and can be cleared by setting the CLR control input.
- The Accumulator’s input must come via the Adder & Logic Circuit. This allows the Accumulator and Data Register to swap data simultaneously.
- The address of any memory location being accessed must be loaded in the Address Register.

Basic Computer Registers Connected to a Common Bus

Nb: All except INPR and Adder are connected to a clock pulse

- **Bus**
  - $S_2$ $S_1$ $S_0$

- **Memory** 4096 x 16
  - Write
  - Read
  - **AR**
    - LD INR CLR
  - **PC**
    - LD INR CLR

- **Register**
  - **DR**
    - LD INR CLR
  - **AC**
    - LD INR CLR

- **INPR**
  - LD

- **IR**
  - LD

- **TR**
  - LD INR CLR

- **OUTR**

- **E**
  - Adder & Logic

Diagrams for the registers and connections are shown in the area marked "Basic Computer Registers Connected to a Common Bus."
Computer Instructions

- The basic computer has three instruction code formats:
  - **Memory-reference format** – where seven 3-bit opcodes are followed by a 12-bit memory address and preceded by a bit which indicates whether direct or indirect addressing is being used.
  - **Register-reference format** – where 0111₂ is followed by 12 bits which indicate a register instruction.
  - **Input-output format** – where 1111₂ is followed by 12 bit which indicate an input-output instruction.
- In register-reference and I/O formats, only one of the lower 12 bits is set.
Instruction-Set Completeness

• A computer instruction set is said to be complete if the computer includes a sufficient number of instructions in each of these categories:
  – Arithmetic, logical and shift instructions
  – Instructions for moving data from registers to memory and memory to registers.
  – Program-control and status-checking instructions
  – Input and output instructions

Arithmetic, Logic and Shifting Completeness

• We have instructions for adding, complementing and incrementing the accumulator. With these we can also subtract.
• AND and complement provide NAND, from which all other logical operations can be constructed.
• We can construct logical and arithmetic shifts from the circular shift operations.
• We can construct multiply and divide from adding, subtracting and shifting.
• While this is complete, it is not very efficient; it would be to our advantage to have subtract, multiply, OR and XOR.
Instruction Set Completeness (continued)

- We can perform moves using the LDA and STA instructions.
- We have unconditional branches (BUN), subprogram calls (BSA) and conditional branches (ISZ).
- We also have all the instructions we need to perform input and output and handle the interrupt that they generate.

### Basic Memory-Reference Instructions

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Hexadecimal code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>0xxx 8xxx</td>
<td>AND mem. Word to AC</td>
</tr>
<tr>
<td>ADD</td>
<td>1xxx 9xxx</td>
<td>ADD mem. Word to AC</td>
</tr>
<tr>
<td>LDA</td>
<td>2xxx Axxx</td>
<td>Load mem. Word to AC</td>
</tr>
<tr>
<td>STA</td>
<td>3xxx Bxxx</td>
<td>Store Content of AC in mem.</td>
</tr>
<tr>
<td>BUN</td>
<td>4xxx Cxxx</td>
<td>Branch unconditionally</td>
</tr>
<tr>
<td>BSA</td>
<td>5xxx Dxxx</td>
<td>Branch and save return address</td>
</tr>
<tr>
<td>ISZ</td>
<td>6xxx Exxx</td>
<td>Increment and skip if zero</td>
</tr>
</tbody>
</table>
# Basic Register-Reference Instructions

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Hex. Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLA</td>
<td>7800</td>
<td>Clear AC</td>
</tr>
<tr>
<td>CLE</td>
<td>7400</td>
<td>Clear E</td>
</tr>
<tr>
<td>CMA</td>
<td>7200</td>
<td>Complement AC</td>
</tr>
<tr>
<td>CME</td>
<td>7100</td>
<td>Complement E</td>
</tr>
<tr>
<td>CIR</td>
<td>7080</td>
<td>Circulate right AC &amp; E</td>
</tr>
<tr>
<td>CIL</td>
<td>7040</td>
<td>Circulate left AC &amp; E</td>
</tr>
<tr>
<td>INC</td>
<td>7020</td>
<td>Increment AC</td>
</tr>
</tbody>
</table>

# Basic Register-Reference Instructions (continued)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Hex. Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPA</td>
<td>7010</td>
<td>Skip next instruction if AC is positive</td>
</tr>
<tr>
<td>SNA</td>
<td>7008</td>
<td>Skip next instruction if AC is negative</td>
</tr>
<tr>
<td>SZA</td>
<td>7004</td>
<td>Skip next instruction if AC is zero</td>
</tr>
<tr>
<td>SZE</td>
<td>7002</td>
<td>Skip next instruction if E is zero</td>
</tr>
<tr>
<td>HLT</td>
<td>7001</td>
<td>Halt computer</td>
</tr>
</tbody>
</table>
### Basic Input-Output Instructions

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Hex. Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INP</td>
<td>F800</td>
<td>Input character to AC</td>
</tr>
<tr>
<td>OUT</td>
<td>F400</td>
<td>Output character from AC</td>
</tr>
<tr>
<td>SKI</td>
<td>F200</td>
<td>Skip on input flag</td>
</tr>
<tr>
<td>SKO</td>
<td>F100</td>
<td>Skip on output flag</td>
</tr>
<tr>
<td>ION</td>
<td>F080</td>
<td>Interrupt on</td>
</tr>
<tr>
<td>IOF</td>
<td>F040</td>
<td>Interrupt off</td>
</tr>
</tbody>
</table>

### Timing and Control

- The timings for all the registers is controlled a master clock generator.
  - Its pulses are applied to all flip-flops and registers, including in the control unit.
  - The control signals are generated in the control unit and provide control inputs for the bus’s multiplexers and for the processor registers and provides microoperations for the accumulator.
Control

- There are two types of control:
  - **Hardwired** – control logic is implemented with gates, flip-flops, decoders and other digital circuits.
  - **Microprogrammed** – control information is stored in a control program, which is programmed to perform the necessary steps to implement instructions.

Timing Signals

- Timing signals are generated by the sequence counter (SC), which receives as inputs the clock pulse, increment and clear.
- The SC’s outputs are decoded into 16 timing signal T₀ through T₁₅, which are used to control the sequence of operations.
- The RTL statement
  \[ D_3T_4: \ SC \leftarrow 0 \]
  resets the sequence counter to zero; the next timing signal is T₀
Control Unit of Basic Computer

Instruction Register (IR)

Other inputs

Control Logic Gates

Control outputs

Examples of Control Timing Signals
Instruction Cycle

- The instructions of a program are carried out by a process called the **instruction cycle**.
- The instruction cycle consists of these phases:
  - Fetch an instruction from memory
  - Decode the instruction
  - Read the effective address from memory if the operand has an indirect address.
  - Execute the instruction.

Fetch and Decode

- Initially, the PC has stored the address of the instruction about to be executed and the SC is cleared to 0.
- With each clock pulses the SC is incremented and the timing signals go through the sequence $T_0$, $T_1$, $T_2$, etc.
- It is necessary to load the AR with the PC’s address (it is connected to memory address inputs):
  $$T_0: \text{AR} \leftarrow \text{PC}$$
Fetch and Decode

- Subsequently, as we fetch the instruction to be executed, we must increment the program counter so that it points to the next instruction:
  - $T_1$: $\text{IR} \leftarrow \text{M}[\text{AR}], \text{PC} \leftarrow \text{PC} + 1$
- In order to carry out the instruction, we must decode and prepare to fetch the operand. In the event it is an indirect operand, we need to have the indirect addressing bit as well:
  - $T_2$: $D_0, \ldots, D_7 \leftarrow \text{Decode IR(12-14)},
    \text{AR} \leftarrow \text{IR (0-11)}, I \leftarrow \text{IR(15)}$

Register Transfers For the Fetch Phase
Type of Instruction and Addressing

- During time $T_3$, the control unit determines if this is a memory-reference, register-reference or input/output instruction.
  - The latter two are distinguished by the I (indirect) bit.
  - If it is a memory-reference instruction, the I bit will determine direct or indirect addressing.
- The four separate paths are:
  - $D_7 IT_3$: $AR \leftarrow M[AR]$
  - $D_7 IT_3$: Nothing
  - $D_7 IT_3$: Execute a register-reference instruction
  - $D_7 IT_3$: Execute an input-output instruction

Flowchart For Instruction Cycle
Execution of Register-Reference Instructions

D_1T_3 = r (common to all register-reference instructions)
IR(i) = B_i [bit in IR(0-11) that specifies the operation]

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>SC ← 0</td>
</tr>
<tr>
<td>CLA rB_11</td>
<td>AC ← 0</td>
</tr>
<tr>
<td>CLE rB_10</td>
<td>E ← 0</td>
</tr>
<tr>
<td>CMA rB_9</td>
<td>AC ← AC'</td>
</tr>
<tr>
<td>CME rB_8</td>
<td>E ← E'</td>
</tr>
<tr>
<td>CIR rB_7</td>
<td>AC ← shr AC, E ← AC(15)</td>
</tr>
<tr>
<td>CIL rB_6</td>
<td>AC ← shl AC, E ← AC(0)</td>
</tr>
<tr>
<td>INC rB_5</td>
<td>AC ← AC + 1</td>
</tr>
</tbody>
</table>

Execution of Register-Reference Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPA rB_4</td>
<td>If (AC(15) = 0) then PC ← PC + 1</td>
<td>Skip if positive</td>
</tr>
<tr>
<td>SNA rB_3</td>
<td>If (AC(15) = 1) then PC ← PC + 1</td>
<td>Skip if negative</td>
</tr>
<tr>
<td>SZA rB_2</td>
<td>If (AC = 0) then PC ← PC + 1</td>
<td>Skip if AC zero</td>
</tr>
<tr>
<td>SZE rB_1</td>
<td>If (E = 0) then PC ← PC + 1</td>
<td>Skip if E zero</td>
</tr>
<tr>
<td>HLT rB_0</td>
<td>S ← 0 (S is a start-stop flip-flop)</td>
<td>Halt computer</td>
</tr>
</tbody>
</table>
Memory-Reference Instructions

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>D₀</td>
<td>AC ← AC ∧ M[AR]</td>
</tr>
<tr>
<td>ADD</td>
<td>D₁</td>
<td>AC ← AC + M[AR], E ← C_out</td>
</tr>
<tr>
<td>LDA</td>
<td>D₂</td>
<td>AC ← M[AR]</td>
</tr>
<tr>
<td>STA</td>
<td>D₃</td>
<td>M[AR] ← AC</td>
</tr>
<tr>
<td>BUN</td>
<td>D₄</td>
<td>PC ← AR</td>
</tr>
<tr>
<td>BSA</td>
<td>D₅</td>
<td>M[AR] ← PC, PC ← AR + 1</td>
</tr>
<tr>
<td>ISZ</td>
<td>D₆</td>
<td>M[AR] ← M[AR] + 1, If M[AR] + 1 = 0 then PC ← PC + 1</td>
</tr>
</tbody>
</table>

- All memory-reference instructions have to wait until $T₄$ so that the timing is the same whether the operand is direct or indirect.
- **AND, ADD** and **LDA** must all be performed in two steps because AC can only be access via DR:
  - **AND**: $D₀T₄$: DR ← M[AR]  
    $D₀T₅$: AC ← AC ∧ DR, SC ← 0  
  - **ADD**: $D₁T₄$: DR ← M[AR]  
    $D₁T₅$: AC ← AC + DR, E ← C_out, SC ← 0  
  - **LDA**: $D₂T₄$: DR ← M[AR]  
    $D₀T₅$: AC ← DR, SC ← 0
Memory-Reference Instructions (continued)

- **STA** stores the contents of the AC, which can be applied directly to the bus:
  
  \[ D_3T_4: \quad M[AR] \leftarrow AC, \quad SC \leftarrow 0 \]

- **BUN** transfers control unconditionally to the effective address indicated by the effective address:
  
  \[ D_4T_4: \quad PC \leftarrow AR, \quad SC \leftarrow 0 \]

- **BSA** is used to branch to a subprogram. This requires saving the return address, which is saved at the operand’s effective address with the subprogram beginning one word later in memory:
  
  \[ D_5T_4: \quad M[AR] \leftarrow PC, \quad AR \leftarrow AR + 1 \]

  \[ D_5T_5: \quad PC \leftarrow AR, \quad SC \leftarrow 0 \]

---

**Example of BSA Instruction Execution**

<table>
<thead>
<tr>
<th>Memory</th>
<th>20</th>
<th>0</th>
<th>BSA</th>
<th>135</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Next instruction</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory, PC, &amp; AR at time ( T_4 )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
<th>20</th>
<th>0</th>
<th>BSA</th>
<th>135</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Next instruction</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory &amp; PC after execution</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Memory-Reference Instructions
(continued)

- **ISZ** skips the next instruction if the operand stored at the effective address is 0. This requires that the PC incremented, which cannot be done directly:
  - $D_6T_4$: \( DR \leftarrow M[AR] \)
  - $D_6T_5$: \( DR \leftarrow DR + 1 \)
  - $D_6T_6$: \( M[AR] \leftarrow DR, \) if \( DR = 0 \) then \( PC \leftarrow PC + 1 \), \( SC \leftarrow 0 \)

Flowchart For Memory-Reference Instructions
Flowchart For Memory-Reference Instructions (continued)

**Memory-reference Instructions**

- **BUN**
  - PC ← AR
  - SC ← 0
  - D₄₁₄

- **BSA**
  - M[AR] ← PC
  - AR ← AR + 1
  - D₄₁₅

- **ISZ**
  - DR ← M[AR]
  - DR ← DR + 1
  - D₄₆₆

IF (DR = 0)
then (PC ← PC + 1)
SC ← 0

Input-Output Configuration

**Input-output terminal**
- Printer
- Keyboard

**Serial Comm. Interface**
- Receiver Interface
- Transmitter Interface

**Comp. Registers and Flip-flops**
- FGO = 1 NOP
- FGO = 0 output data
- OUTR
- INPR = 0 NOP
- INPR = 1 input waiting
- FGI
Input-Output Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INP pB&lt;sup&gt;11&lt;/sup&gt;</td>
<td>AC(0-7) ← INPR, FGI ← 0, Input character</td>
</tr>
<tr>
<td>OUT pB&lt;sup&gt;10&lt;/sup&gt;</td>
<td>OUTR ← AC(0-7), FGO ← 0, Output character</td>
</tr>
<tr>
<td>SKI pB&lt;sup&gt;9&lt;/sup&gt;</td>
<td>If (FGI = 1) Then PC ← PC + 1, Skip on input flag</td>
</tr>
<tr>
<td>SKO pB&lt;sup&gt;8&lt;/sup&gt;</td>
<td>If (FGO = 1) Then PC ← PC + 1, Skip on output flag</td>
</tr>
<tr>
<td>ION pB&lt;sup&gt;7&lt;/sup&gt;</td>
<td>IEN ← 1, Interrupt enable on</td>
</tr>
<tr>
<td>IOF pB&lt;sup&gt;6&lt;/sup&gt;</td>
<td>IEN ← 0, Interrupt enable off</td>
</tr>
</tbody>
</table>

Clear SC

Flowchart For Interrupt Cycle

Instruction cycle

R = 0

Execute instruction

IEN = 0

R = 1

IEN = 1

FGI = 0

= 1

= 1

= 1

= 1

Branch to location 1

PC ← 1

IEN ← 0

R ← 0

Interrupt cycle

Store return address in location 0

M[0] ← PC
Demonstration of the Interrupt Cycle

Before Interrupt

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>0 BUN 1120</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

PC = 255
256
1120

Main Program

I/O Program

1 BUN 0

After Interrupt Cycle

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>0 BUN 1120</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

PC = 1
255
256
1120

Main Program

I/O Program

1 BUN 0

Flowchart For Computer Operation

Start

SC ← 0, IEN ← 0, R ← 0

Instruction Cycle

R'T₀ = 0

AR ← PC

R'T₁

IR ← M[AR], PC ← PC + 1

R'T₂

AR ← IR(0-11), I ← IR(15)
D₀...D₇ ← Decode IR(12-14)

Interrupt Cycle

R'T₀ = 1

AR ← 0, TR ← PC

R'T₁

M[AR] ← TR, PC ← 0

R'T₂

PC ← PC + 1, IEN ← 0
R ← 0, SC ← 0

D₀JT₁ = 1

 Execute I/O Instruction

D₁JT₁ = 0

 Execute Reg. Instruction

D₂JT₁ = 1

 Execute Mem. Instruction

D₃JT₁ = 0

 Nothing
Control Gates Associated With AR

Control Inputs for IEN
### Encoder for Bus Selection Inputs

![Diagram of Encoder for Bus Selection Inputs]

### Encoder for Bus Selection Circuit

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
<th>Register Selected for Bus</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td>$x_2$</td>
<td>$x_3$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Circuits Associated With AC

Control Gates

Adder & Logic Circuit

Accumulator Register (AC)

16 From DR
16 From INPR

16 To bus

LD INR CLR Clock

Gate structure for Controlling the LD, INR, CLR of AC

D0 T5
D1
D2 T3
p B11
r B9
B7
B6
B5
B4
B3
B2
B11

AND
ADD
DR
INPR
COM
SHR
SHL
INC
CLR

16 From adder and logic
16 To bus

Clock

AC
One stage of Adder and Logic Circuit