Lecture 5: Computer Architecture

- Simpletron
  - Simpletron Architecture
  - Simpletron Instruction Set
  - Example Programs

The Simpletron is described in *C How to Program* by Harvey Deitel.

- Computer Architecture
  - Registers
  - Flags
  - Address Calculation

Simpletron Architecture

CPU
- The CPU contains one general-purpose register called the accumulator

Memory
- All information in the Simpletron is handled in terms of words. A word is a signed four-digit decimal number such as +3364 or -0001
- The Simpletron is equipped with a 100-word memory, and these words are referenced by their location numbers 00-99
- Before running a Simpletron Machine Language (SML) program, it must be loaded into memory. The first instruction of every SML program is always placed in location 00.
- Each instruction occupies one word of memory. The sign of an SML instruction is always positive, but the sign of a data word may be either positive or negative. Each location in the Simpletron’s memory may contain either an instruction, a data value used by the program, or an unused area of memory.

I/O
- The Simpletron uses a keyboard for input and a terminal screen for output

Simpletron

Instruction Set

<table>
<thead>
<tr>
<th>Instruction Op code Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ 10 Reads a word from the terminal into a specific location in memory</td>
</tr>
<tr>
<td>WRITE 11 Writes a word from a specific location in memory to the terminal</td>
</tr>
<tr>
<td>LOAD 20 Loads a word from a specific location in memory into the accumulator</td>
</tr>
<tr>
<td>STORE 21 Stores a word from the accumulator into a specific location in memory</td>
</tr>
<tr>
<td>ADD 30 Adds a word from a specific location in memory to the word in the accumulator (leaves result in the accumulator)</td>
</tr>
<tr>
<td>SUBTRACT 31 Subtracts a word from a specific location in memory from the word in the accumulator (leaves result in the accumulator)</td>
</tr>
</tbody>
</table>
Instruction Set (cont.)

<table>
<thead>
<tr>
<th>Instruction Op code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRANCH 40</td>
<td>Branches to a specific location in memory</td>
</tr>
<tr>
<td>BRANCHNEG 41</td>
<td>Branches to a specific location in memory if the accumulator is negative</td>
</tr>
<tr>
<td>BRANCHZERO 42</td>
<td>Branches to a specific location in memory if the accumulator is zero</td>
</tr>
<tr>
<td>HALT 43</td>
<td>Halts</td>
</tr>
</tbody>
</table>

Simple Program

- Read two numbers, add them together, and print the sum.
- Algorithm:
  - read A
  - read B
  - sum = A + B
  - print sum
  - stop

Assembly Instructions

- read A  (reads into the memory location used to store A)
- read B  (puts A into the accumulator)
- load A  (adds B to the accumulator, leaving the result in the accumulator)
- add B   (saves the number in the accumulator into memory)
- store Sum (writes out the result to the terminal)
- write Sum
- halt

Simpletron Assembly Language → Simpletron Machine Language

- 1-1 Translation SAL->SML
- Execution starts at location 0
- In our example:
  - 7 instructions : locations 00 – 06
  - 3 data values
- Where to put data?
  - Directly after the program, or
  - In high memory, working down (99 and lower)
- (homework hint: if you use the second option, then if you need to add instructions you can leave the data where it is and not have to re-do all your machine language that refers to it!)
Translating into Simpletron Machine Code

- First, determine which memory locations you want to use for your data:
  - 97 – A
  - 98 – B
  - 99 – Sum

- Then, look up the opcodes for each instruction
  - 1097 read A (10 = read, 97 = location of A)
  - 1098 read B (10 = read, 98 = B location)
  - 2097 load A (20 = load, 97 = A location))
  - 3098 add B (30 = add, 98 = B location)

Translating, cont.

- 2199 store Sum (21 = store, 99 = sum location)
- 1199 write Sum (11 = write, 99 = sum location)
- 4300 halt (43 = halt)

Program Execution

<table>
<thead>
<tr>
<th>Location</th>
<th>Contents</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>+1009</td>
<td>Read A</td>
</tr>
<tr>
<td>01</td>
<td>+1010</td>
<td>Read B</td>
</tr>
<tr>
<td>02</td>
<td>+2009</td>
<td>Load A</td>
</tr>
<tr>
<td>03</td>
<td>+3110</td>
<td>Subtract B</td>
</tr>
<tr>
<td>04</td>
<td>+4107</td>
<td>If B &gt; A, go to 07</td>
</tr>
<tr>
<td>05</td>
<td>+1109</td>
<td>Write A</td>
</tr>
<tr>
<td>06</td>
<td>+4300</td>
<td>Halt</td>
</tr>
<tr>
<td>07</td>
<td>+1110</td>
<td>Write B</td>
</tr>
<tr>
<td>08</td>
<td>+4300</td>
<td>Halt</td>
</tr>
<tr>
<td>09</td>
<td>+0000</td>
<td>(Variable A)</td>
</tr>
<tr>
<td>10</td>
<td>+0000</td>
<td>(Variable B)</td>
</tr>
</tbody>
</table>

Branching Example

Read two numbers from the keyboard and print the larger value:

<table>
<thead>
<tr>
<th>Location</th>
<th>Contents</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>+1009</td>
<td>Read A</td>
</tr>
<tr>
<td>01</td>
<td>+1010</td>
<td>Read B</td>
</tr>
<tr>
<td>02</td>
<td>+2009</td>
<td>Load A</td>
</tr>
<tr>
<td>03</td>
<td>+3110</td>
<td>Subtract B</td>
</tr>
<tr>
<td>04</td>
<td>+4107</td>
<td>If B &gt; A, go to 07</td>
</tr>
<tr>
<td>05</td>
<td>+1109</td>
<td>Write A</td>
</tr>
<tr>
<td>06</td>
<td>+4300</td>
<td>Halt</td>
</tr>
<tr>
<td>07</td>
<td>+1110</td>
<td>Write B</td>
</tr>
<tr>
<td>08</td>
<td>+4300</td>
<td>Halt</td>
</tr>
<tr>
<td>09</td>
<td>+0000</td>
<td>(Variable A)</td>
</tr>
<tr>
<td>10</td>
<td>+0000</td>
<td>(Variable B)</td>
</tr>
</tbody>
</table>
Looping Example

Use a loop to print the numbers one through 10:

<table>
<thead>
<tr>
<th>LOCATION</th>
<th>CONTENTS</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>+1107</td>
<td>Write the value of the variable Number</td>
</tr>
<tr>
<td>01</td>
<td>+2007</td>
<td>Load Number into the accumulator</td>
</tr>
<tr>
<td>02</td>
<td>+3008</td>
<td>Increment the accumulator by 1</td>
</tr>
<tr>
<td>03</td>
<td>+2107</td>
<td>Store incremented value back in Number</td>
</tr>
<tr>
<td>04</td>
<td>+3109</td>
<td>Subtract 11 from accumulator</td>
</tr>
<tr>
<td>05</td>
<td>+4100</td>
<td>Go to 00 if 10 iterations haven't been completed</td>
</tr>
<tr>
<td>06</td>
<td>+4300</td>
<td>All done; Halt</td>
</tr>
<tr>
<td>07</td>
<td>+0001</td>
<td>(Number)</td>
</tr>
<tr>
<td>08</td>
<td>+0001</td>
<td>Constant 1 (used for incrementing)</td>
</tr>
<tr>
<td>09</td>
<td>+0011</td>
<td>Constant 11 (loop limit)</td>
</tr>
</tbody>
</table>

Computer Organization

Components

- Control Unit – fetches instructions, decodes instructions, causes instructions to be carried out.
- Arithmetic logical unit (ALU) – performs arithmetic operations (addition, etc.) on data.
- Registers – high speed memory cells (don’t need to go through the bus to access). They vary in number and purpose on different machines.
- Buses – communication pathways connecting different devices/components.
Registers

- 8, 16, or 32 bit high-speed storage locations inside the CPU
- They can be accessed at a much higher speed than conventional memory.
- When optimizing for speed, use registers.
- Four types: general purpose, segment, index, status, and control

General Purpose Registers

- Data registers, also known as general purpose registers: AX, BX, CX, DX
- Used for arithmetic operations and data movement
- Can be addressed as 16 bit or 8 bit values. For AX, upper 8 bits are AH, lower 8 bits are AL.
- Remember: when when a 16 bit register is modified, so is the corresponding 8 bit registers!

Example

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>09</th>
<th>08</th>
<th>07</th>
<th>06</th>
<th>05</th>
<th>04</th>
<th>03</th>
<th>02</th>
<th>01</th>
<th>00</th>
</tr>
</thead>
<tbody>
<tr>
<td>AX</td>
<td>AH</td>
<td>AL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>07</td>
<td>07</td>
<td>00</td>
<td>00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Special Attributes of GP Registers

- AX – accumulator – fastest for arithmetic operations. Some math instructions only use AX.
- BX – base – this register can hold an address of a procedure or variable. BX can also perform arithmetic and data movement.
- CX – counter – this register acts as a counter for repeating or looping instructions
- DX – data – this register has a special role in multiply and divide operations. In multiplication it holds the high 16 bits of the product. In division it holds the remainder.
Segment Registers

- Segment registers are used as base locations for program instructions, data, and the stack.
- All references to memory involve a segment register as the base location.

Segment Registers, cont.

- CS – code segment – this register holds the base location of all instructions in a program
- DS – data segment – this is the default base location for variables. It is used by the CPU to calculate the variable location.
- SS – stack segment – this register contains the base location of the stack.
- ES – extra segment – this is an additional base location for memory variables.

Index Registers

- Index registers contain the offsets of data and instructions.
- Offset refers to the distance of a variable, label, or instruction from its base segment.
- Index registers are used when processing strings, arrays, and other data structures.

Index Registers, cont.

- BP – base pointer – this register contains an offset from the SS register and is often used by subroutines to find the variables passed to it on the stack.
- SP – stack pointer – this register contains the offset from the top of the stack. The complete top of stack address is calculated using the SP and SS registers.
- SI – source index – used to point to data in memory. Named because this is the index register commonly used as the source in string operations (for example)
- DI - destination index – index register commonly used as the destination in string operations
Status and Control Registers

- IP – instruction pointer – always contains the offset of the next instruction. The IP and CS registers combine to form the complete address. IP is also known as PC – the program counter.
- Flags – a special register with individual bit positions that give the status of the CPU (control flags) or results of arithmetic operations (status flags).

Status Flags

- These indicate the status of arithmetic and logical operations.
- Carry flag (CF) – set if the result of an unsigned operation is too big to fit into the destination. 1 = carry, 0 = no carry.
- Overflow flag (OF) – set if the result of a signed operation is too wide to fit into the destination. 1 = overflow, 0 = no overflow.
- Sign flag (SF) – set when the result of an operation is negative. 1 = negative, 0 = positive

Status Flags, cont.

- Zero flag (ZF) – set when the result of an arithmetic operation is zero. Used by branch and loop instructions when comparing values. 1 = zero, 0 = not zero.
- Auxiliary Carry – set when an operation causes a carry from bit 3 to bit 4 or a borrow from bit 4 to bit 3. 1 = carry, 0 = no carry.
- Parity – indicates if the result of an operation has an even or odd number of bits. Used to verify memory integrity or correct transmission of data.

Addressing

- Address: a number referring to an 8-bit memory location
- Logical addresses go from 0 to the highest location
- How these are translated into physical addresses varies.
- For Intel:
  - 32-bit segment-offset address: combination of base location (segment) and offset to represent a logical location
  - 20-bit absolute address, which refers to a physical memory location
Addressing, cont.

- Problem: how to address 1,048,576 bytes of memory with a 16-bit wide address register (where the max is 65,535)
- Solution: combine segment and offset values to obtain the absolute address
- Example: 08F1:0100
  1) convert segment to absolute by adding 4 zero bits: 08F10
  2) add the offset: 0100 (hex)
     \[ 08F10 \rightarrow \text{segment value w/extra 4 0 bits} \]
     \[ + 0100 \rightarrow \text{add the offset} \]
     \[ 09010 \rightarrow \text{obtain the absolute address} \]
     (effective address)

Why Segment-Offset?

- You can load the program at any segment address and individual variable addresses to not need to be recalculated.
  - Why? Variable locations are 16-bit offsets from the program’s data area.
  - This is known as being segment relocatable.
- Programs can access large data structures by modifying the segment portion of the data’s address to point to new blocks of memory.

Data Segment

addressable memory on 8086

Example:
- DS = 0100h
data seg. start?
data seg. end?
- If BX contains the offset:

  \[ \text{BX} = 005Ah \]

  \[ \text{EA} = ? \]

Segment Register Combinations

- Code Segment – the CS register and IP (instruction pointer) are used to point to the next instruction.
- Stack – the SS register is used with the SP (stack pointer) or BP (base pointer)
- Data Segment – DS with BX, SI, or DI
- Extended Segment – BX, SI, or DI
More on Effective Addresses

• There’s more than one way to get the same effective address!
• Example:
  – CS = 147Bh
  – IP = 131Ah
  – EA = 147B0 + 131A = 15ACAh

  or
  – CS = 15ACH
  – IP = 000Ah
  – EA = 15AC + 000A = 15ACAh

• If CS = 147B, what range of effective addresses can be referenced without changing the value in CS?