Chapter 4 – Register Transfer and Microoperations

Section 4.1 – Register Transfer Language

- Digital systems are composed of modules that are constructed from digital components, such as registers, decoders, arithmetic elements, and control logic
- The modules are interconnected with common data and control paths to form a digital computer system
- The operations executed on data stored in registers are called microoperations
- A microoperation is an elementary operation performed on the information stored in one or more registers
- Examples are shift, count, clear, and load
- Some of the digital components from before are registers that implement microoperations
- The internal hardware organization of a digital computer is best defined by specifying
  - The set of registers it contains and their functions
  - The sequence of microoperations performed on the binary information stored
  - The control that initiates the sequence of microoperations
- Use symbols, rather than words, to specify the sequence of microoperations
- The symbolic notation used is called a register transfer language
- A programming language is a procedure for writing symbols to specify a given computational process
- Define symbols for various types of microoperations and describe associated hardware that can implement the microoperations

Section 4.2 – Register Transfer

- Designate computer registers by capital letters to denote its function
- The register that holds an address for the memory unit is called MAR
- The program counter register is called PC
- IR is the instruction register and R1 is a processor register
- The individual flip-flops in an $n$-bit register are numbered in sequence from 0 to $n-1$
- Refer to Figure 4.1 for the different representations of a register
Designate information transfer from one register to another by

\[ R2 \leftarrow R1 \]

This statement implies that the hardware is available
- The outputs of the source must have a path to the inputs of the destination
- The destination register has a parallel load capability

If the transfer is to occur only under a predetermined control condition, designate it by

\[ \text{If } (P = 1) \text{ then } (R2 \leftarrow R1) \]

or,

\[ P: R2 \leftarrow R1, \]

where \( P \) is a control function that can be either 0 or 1

Every statement written in register transfer notation implies the presence of the required hardware construction
It is assumed that all transfers occur during a clock edge transition.

All microoperations written on a single line are to be executed at the same time:

\[ T: R2 \leftarrow R1, R1 \leftarrow R2 \]

---

**TABLE 4-1 Basic Symbols for Register Transfers**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Letters</td>
<td>Denotes a register</td>
<td>( MAR, R2 )</td>
</tr>
<tr>
<td>(and numerals)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parentheses</td>
<td>Denotes a part of a register</td>
<td>( R2(0-7), R2(L) )</td>
</tr>
<tr>
<td>Arrow ( \leftarrow )</td>
<td>Denotes transfer of information</td>
<td>( R2 \leftarrow R1 )</td>
</tr>
<tr>
<td>Comma ,</td>
<td>Separates two microoperations</td>
<td>( R2 \leftarrow R1, R1 \leftarrow R2 )</td>
</tr>
</tbody>
</table>
Section 4.3 – Bus and Memory Transfers

- Rather than connecting wires between all registers, a common bus is used
- A bus structure consists of a set of common lines, one for each bit of a register
- Control signals determine which register is selected by the bus during each transfer
- Multiplexers can be used to construct a common bus
- Multiplexers select the source register whose binary information is then placed on the bus
- The select lines are connected to the selection inputs of the multiplexers and choose the bits of one register

In general, a bus system will multiplex $k$ registers of $n$ bits each to produce an $n$-line common bus
- This requires $n$ multiplexers – one for each bit
- The size of each multiplexer must be $k \times 1$
- The number of select lines required is $\log k$
- To transfer information from the bus to a register, the bus lines are connected to the inputs of all destination registers and the corresponding load control line must be activated
- Rather than listing each step as
Instead of using multiplexers, *three-state gates* can be used to construct the bus system.

- A three-state gate is a digital circuit that exhibits three states.
- Two of the states are signals equivalent to logic 1 and 0.
- The third state is a *high-impedance* state – it behaves like an open circuit, which means the output is disconnected and does not have a logic significance.

**Figure 4-4** Graphic symbols for three-state buffer.

- The three-state buffer gate has a normal input and a control input which determines the output state.
- With control 1, the output equals the normal input.
- With control 0, the gate goes to a high-impedance state.
- This enables a large number of three-state gate outputs to be connected with wires to form a common bus line without endangering loading effects.
• Decoders are used to ensure that no more than one control input is active at any given time
• This circuit can replace the multiplexer in Figure 4.3
• To construct a common bus for four registers of $n$ bits each using three-state buffers, we need $n$ circuits with four buffers in each
• Only one decoder is necessary to select between the four registers

• Designate a memory word by the letter M
• It is necessary to specify the address of M when writing memory transfer operations
• Designate the address register by AR and the data register by DR
• The read operation can be stated as:
  \[
  \text{Read: } DR \leftarrow M[AR]
  \]
• The write operation can be stated as:
  \[
  \text{Write: } M[AR] \leftarrow R1
  \]

Section 4.4 – Arithmetic Microoperations

• There are four categories of the most common microoperations:
  o Register transfer: transfer binary information from one register to another
  o Arithmetic: perform arithmetic operations on numeric data stored in registers
- Logic: perform bit manipulation operations on non-numeric data stored in registers
- Shift: perform shift operations on data stored in registers

- The basic arithmetic microoperations are addition, subtraction, increment, decrement, and shift
- Example of addition: \( R3 \leftarrow R1 + R2 \)
- Subtraction is most often implemented through complementation and addition
- Example of subtraction: \( R3 \leftarrow R1 + \bar{R2} + 1 \) (strikethrough denotes bar on top – 1’s complement of R2)
- Adding 1 to the 1’s complement produces the 2’s complement
- Adding the contents of R1 to the 2’s complement of R2 is equivalent to subtracting

- Multiply and divide are not included as microoperations
- A microoperation is one that can be executed by one clock pulse
- Multiply (divide) is implemented by a sequence of add and shift microoperations (subtract and shift)

- To implement the add microoperation with hardware, we need the registers that hold the data and the digital component that performs the addition
- A full-adder adds two bits and a previous carry
• A binary adder is a digital circuit that generates the arithmetic sum of two binary numbers of any length
• A binary added is constructed with full-adder circuits connected in cascade
• An n-bit binary adder requires n full-adders

![4-bit binary adder diagram](image)

Figure 4-6 4-bit binary adder.

• The subtraction A-B can be carried out by the following steps
  o Take the 1’s complement of B (invert each bit)
  o Get the 2’s complement by adding 1
  o Add the result to A
• The addition and subtraction operations can be combined into one common circuit by including an XOR gate with each full-adder

![4-bit adder-subtractor diagram](image)

Figure 4-7 4-bit adder-subtractor.

• The increment microoperation adds one to a number in a register
• This can be implemented by using a binary counter – every time the count enable is active, the count is incremented by one
• If the increment is to be performed independent of a particular register, then use half-adders connected in cascade
- An \( n \)-bit binary incrementer requires \( n \) half-adders

- Each of the arithmetic microoperations can be implemented in one composite arithmetic circuit
- The basic component is the parallel adder
- Multiplexers are used to choose between the different operations
- The output of the binary adder is calculated from the following sum:
  \[ D = A + Y + C_{in} \]
Figure 4-9 4-bit arithmetic circuit.
Section 4.5 – Logic Microoperations

- Logic operations specify binary operations for strings of bits stored in registers and treat each bit separately
- Example: the XOR of R1 and R2 is symbolized by
  \[ P: R1 \leftarrow R1 \oplus R2 \]
- Example: R1 = 1010 and R2 = 1100
  
  \[
  \begin{array}{c|c|c|c|c|c}
  & 1010 & 1100 & 0110 \\
  \hline
  \text{Content of R1} & \text{Content of R2} & \text{Content of R1 after } P = 1 \\
  \hline
  
  \end{array}
  \]

- Symbols used for logical microoperations:
  - OR: \( \lor \)
  - AND: \( \land \)
  - XOR: \( \oplus \)

- The + sign has two different meanings: logical OR and summation
- When + is in a microoperation, then summation
- When + is in a control function, then OR
- Example:
  \[ P + Q: R1 \leftarrow R2 + R3, R4 \leftarrow R5 \lor R6 \]
- There are 16 different logic operations that can be performed with two binary variables
The hardware implementation of logic microoperations requires that logic gates be inserted for each bit or pair of bits in the registers.

All 16 microoperations can be derived from using four logic gates.

---

**TABLE 4-5** Truth Tables for 16 Functions of Two Variables

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>$F_0$</th>
<th>$F_1$</th>
<th>$F_2$</th>
<th>$F_3$</th>
<th>$F_4$</th>
<th>$F_5$</th>
<th>$F_6$</th>
<th>$F_7$</th>
<th>$F_8$</th>
<th>$F_9$</th>
<th>$F_{10}$</th>
<th>$F_{11}$</th>
<th>$F_{12}$</th>
<th>$F_{13}$</th>
<th>$F_{14}$</th>
<th>$F_{15}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**TABLE 4-6** Sixteen Logic Microoperations

<table>
<thead>
<tr>
<th>Boolean function</th>
<th>Microoperation</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F_0 = 0$</td>
<td>$F \leftarrow 0$</td>
<td>Clear</td>
</tr>
<tr>
<td>$F_1 = xy$</td>
<td>$F \leftarrow A \land B$</td>
<td>AND</td>
</tr>
<tr>
<td>$F_2 = xy'$</td>
<td>$F \leftarrow A \land \overline{B}$</td>
<td></td>
</tr>
<tr>
<td>$F_3 = x$</td>
<td>$F \leftarrow \overline{A}$</td>
<td>Transfer A</td>
</tr>
<tr>
<td>$F_4 = x' y$</td>
<td>$F \leftarrow \overline{A} \land B$</td>
<td></td>
</tr>
<tr>
<td>$F_5 = y$</td>
<td>$F \leftarrow B$</td>
<td>Transfer B</td>
</tr>
<tr>
<td>$F_6 = x \oplus y$</td>
<td>$F \leftarrow A \oplus B$</td>
<td>Exclusive-OR</td>
</tr>
<tr>
<td>$F_7 = x + y$</td>
<td>$F \leftarrow A \lor B$</td>
<td>OR</td>
</tr>
<tr>
<td>$F_8 = (x + y)'$</td>
<td>$F \leftarrow \overline{A} \lor \overline{B}$</td>
<td>NOR</td>
</tr>
<tr>
<td>$F_9 = (x \oplus y)'$</td>
<td>$F \leftarrow \overline{A} \oplus \overline{B}$</td>
<td>Exclusive-NOR</td>
</tr>
<tr>
<td>$F_{10} = y'$</td>
<td>$F \leftarrow \overline{B}$</td>
<td>Complement B</td>
</tr>
<tr>
<td>$F_{11} = x + y'$</td>
<td>$F \leftarrow \overline{A} \lor \overline{B}$</td>
<td>Complement A</td>
</tr>
<tr>
<td>$F_{12} = x'$</td>
<td>$F \leftarrow \overline{A}$</td>
<td></td>
</tr>
<tr>
<td>$F_{13} = x' + y$</td>
<td>$F \leftarrow \overline{A} \lor B$</td>
<td></td>
</tr>
<tr>
<td>$F_{14} = (xy)'$</td>
<td>$F \leftarrow \overline{A} \land \overline{B}$</td>
<td>NAND</td>
</tr>
<tr>
<td>$F_{15} = 1$</td>
<td>$F \leftarrow \text{all 1’s}$</td>
<td>Set to all 1’s</td>
</tr>
</tbody>
</table>
• Logic microoperations can be used to change bit values, delete a group of bits, or insert new bit values into a register.

• The *selective-set* operation sets to 1 the bits in A where there are corresponding 1’s in B

\[
\begin{align*}
\text{1010} & \quad \text{A before} \\
\text{1100} & \quad \text{B (logic operand)} \\
\text{1110} & \quad \text{A after}
\end{align*}
\]

\[A \leftarrow A \lor B\]

• The *selective-complement* operation complements bits in A where there are corresponding 1’s in B

\[
\begin{align*}
\text{1010} & \quad \text{A before} \\
\text{1100} & \quad \text{B (logic operand)} \\
\text{0110} & \quad \text{A after}
\end{align*}
\]

\[A \leftarrow A \oplus B\]

• The *selective-clear* operation clears to 0 the bits in A only where there are corresponding 1’s in B

\[
\begin{align*}
\text{1010} & \quad \text{A before} \\
\text{1100} & \quad \text{B (logic operand)} \\
\text{0010} & \quad \text{A after}
\end{align*}
\]
A ← A ∧ B

- The *mask* operation is similar to the selective-clear operation, except that the bits of A are cleared only where there are corresponding 0’s in B

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>1010</td>
<td>1000</td>
</tr>
<tr>
<td>B (logic operand)</td>
<td></td>
</tr>
</tbody>
</table>

- The *insert* operation inserts a new value into a group of bits
- This is done by first masking the bits to be replaced and then Oring them with the bits to be inserted

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>0110 1010</td>
<td>0000 1010</td>
</tr>
<tr>
<td>B (mask)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 1010</td>
<td>0101 1010</td>
</tr>
<tr>
<td>B (insert)</td>
<td></td>
</tr>
</tbody>
</table>

- The *clear* operation compares the bits in A and B and produces an all 0’s result if the two number are equal

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1010</td>
<td>1010</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 A</td>
<td>A ← A ⊕ B</td>
</tr>
</tbody>
</table>

**Section 4.6 – Shift Microoperations**

- Shift microoperations are used for serial transfer of data
- They are also used in conjunction with arithmetic, logic, and other data-processing operations
- There are three types of shifts: logical, circular, and arithmetic
- A *logical shift* is one that transfers 0 through the serial input
- The symbols *shl* and *shr* are for logical shift-left and shift-right by one position
  
  \[ R1 \leftarrow \text{shl} \ R1 \]

- The *circular shift* (aka rotate) circulates the bits of the register around the two ends without loss of information
- The symbols *cil* and *cir* are for circular shift left and right
The arithmetic shift shifts a signed binary number to the left or right.
- To the left is multiplying by 2, to the right is dividing by 2.
- Arithmetic shifts must leave the sign bit unchanged.
- A sign reversal occurs if the bit in \( R_{n-1} \) changes in value after the shift.
- This happens if the multiplication causes an overflow.
- An overflow flip-flop \( V_s \) can be used to detect the overflow.

\[
V_s = R_{n-1} \oplus R_{n-2}
\]

A bi-directional shift unit with parallel load could be used to implement this.
- Two clock pulses are necessary with this configuration: one to load the value and another to shift.
- In a processor unit with many registers it is more efficient to implement the shift operation with a combinational circuit.
- The content of a register to be shifted is first placed onto a common bus and the output is connected to the combinational shifter, the shifted number is then loaded back into the register.
- This can be constructed with multiplexers.

---

**TABLE 4-7** Shift Microoperations

<table>
<thead>
<tr>
<th>Symbolic designation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R \leftarrow \text{shl} \ R )</td>
<td>Shift-left register ( R )</td>
</tr>
<tr>
<td>( R \leftarrow \text{shr} \ R )</td>
<td>Shift-right register ( R )</td>
</tr>
<tr>
<td>( R \leftarrow \text{cil} \ R )</td>
<td>Circular shift-left register ( R )</td>
</tr>
<tr>
<td>( R \leftarrow \text{cir} \ R )</td>
<td>Circular shift-right register ( R )</td>
</tr>
<tr>
<td>( R \leftarrow \text{ashl} \ R )</td>
<td>Arithmetic shift-left ( R )</td>
</tr>
<tr>
<td>( R \leftarrow \text{ashr} \ R )</td>
<td>Arithmetic shift-right ( R )</td>
</tr>
</tbody>
</table>

**Figure 4-11** Arithmetic shift right.
Section 4.7 – Arithmetic Logic Shift Unit

- The arithmetic logic unit (ALU) is a common operational unit connected to a number of storage registers.
- To perform a microoperation, the contents of specified registers are placed in the inputs of the ALU.
- The ALU performs an operation and the result is then transferred to a destination register.
- The ALU is a combinational circuit so that the entire register transfer operation from the source registers through the ALU and into the destination register can be performed during one clock pulse period.

Figure 4-12 4-bit combinational circuit shifter.
Figure 4.13 One stage of arithmetic logic shift unit.
<table>
<thead>
<tr>
<th>$S_3$</th>
<th>$S_2$</th>
<th>$S_1$</th>
<th>$S_0$</th>
<th>$C_{in}$</th>
<th>Operation</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$F = A$</td>
<td>Transfer $A$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>$F = A + 1$</td>
<td>Increment $A$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>$F = A + B$</td>
<td>Addition</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>$F = A + B + 1$</td>
<td>Add with carry</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>$F = A + \overline{B}$</td>
<td>Subtract with borrow</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>$F = A + \overline{B} + 1$</td>
<td>Subtraction</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>$F = A - 1$</td>
<td>Decrement $A$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$F = A$</td>
<td>Transfer $A$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>$\times$</td>
<td>$F = A \land B$</td>
<td>AND</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>$\times$</td>
<td>$F = A \lor B$</td>
<td>OR</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>$\times$</td>
<td>$F = A \oplus B$</td>
<td>XOR</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$\times$</td>
<td>$F = \overline{A}$</td>
<td>Complement $A$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>$\times$</td>
<td>$\times$</td>
<td>$\times$</td>
<td>$F = \text{shr} A$</td>
<td>Shift right $A$ into $F$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>$\times$</td>
<td>$\times$</td>
<td>$\times$</td>
<td>$F = \text{shl} A$</td>
<td>Shift left $A$ into $F$</td>
</tr>
</tbody>
</table>