CpE 213
Wimp51 – Weekend Instructional Micro Processor; an 8051 subset

Purpose

Provide an overview of computer organization by examining a simple, hypothetical computer called the Wimp51. Examine the instruction set, which is a subset of the 8051's instructions. Develop the block diagram of a datapath plus control unit that can execute Wimp51 instructions. Analyze the operation of a simple Von Neuman class computer by examining the sequence of micro-operations that take place during the execution of various Wimp51 instructions. Write simple programs using the Wimp51 instruction set and assembly language. Translate assembly language to binary machine code. Extend the capability of Wimp51 by adding various hardware features.

Wimp51 Programmer's model

Wimp51 uses a subset of the 8051's instruction set. It's name comes from the fact that the designer constructed it over a weekend and it is intended for instructional use, not for heavy-duty computation! The Wimp51 has a simpler structure than the 8051 as might be expected. The programmer sees Wimp51 as a collection of registers. We often refer to this as the *programmer's model* of a computer. The Wimp51 has an eight bit accumulator (A), eight 8-bit registers R0 through R7, a program counter PC, a carry flag (C), and a zero flag (Z). The C bit is a one bit register that holds the carry out of an addition. The Z flag is actually just all of the accumulator bits NOR'ed together. If all of the bits of the accumulator are equal to zero, the Z flag is equal to 1, otherwise equal to 0. Wimp51 requires an external read-only memory in which to store encoded instructions and has address, data, and control buses to communicate with its memory. The most significant bit of a register is numbered 7, while the least significant bit is numbered 0. The Wimp51 programmer's model is summarized in Figure 1.

Wimp51 Instruction Set

Table 1 shows the instruction set for Wimp51. The first column shows a *mnemonic* for the instruction. A mnemonic is intended to make it easier for a human to understand the instruction. It must be translated into a numeric code that can stored in the computer's memory before it can be executed by the computer. The translation process is called *assembly* and is normally performed by a program called an *assembler*. Later on we will use a program called A51 to assemble programs for the 8051 but for now we'll perform the assembly process manually. The second column shows the numeric machine code that corresponds to each instruction in column one. The third column shows the function of each instruction in a shorthand notation called *register transfer notation*. Here we use a notation similar to the VHDL signal assignment statement.

The first instruction, MOV A,#D, can be read as 'move immediate D into A'. The numeric code that is interpreted by the CPU is normally called *machine code.*

Example 1. Mov immediate

The machine code for the MOV A,#D instruction is shown in column two as 01110100 dddddddd. The first byte is called the *opcode* and can be abbreviated as the base 16 (hex) number 0x74. The second byte is an *immediate operand* as indicated by the sharp sign (#) in the mnemonic. The result of executing this two-byte instruction is to place the immediate operand into the accumulator, as indicated by the register transfer description. You can clear the accumulator by executing the instruction MOV A,#0, which would be encoded as 74 00. Since we normally use hexadecimal when writing machine code, we will dispense with the '0x' notation and just assume hexadecimal numbers.

The next instruction adds the carry bit, the immediate operand, and the accumulator, and puts the 9-bit result into the accumulator and the carry bit. Many computers have both an ADD and an ADDC instruction. Only one is required but it is much easier to do extended precision arithmetic if an ADDC is available. It is easy to clear the carry bit if we only want to add two numbers together, and it is much harder to add in the carry bit with just an ADD instruction so the Wimp51 includes the ADDC but not the ADD. ADDC has two so-called *address modes*: immediate, and register direct. The register direct mode means that the operand is in one of the eight registers. ADDC A,Rn adds the carry bit, accumulator, and register n together and puts the result into the carry bit and the accumulator. Six instructions use the register direct mode, while two use immediate mode.

Notice that the encoding for MOV Rn,A is F8 through FF. The address of register operand is included in the instruction. In this case, the opcode is only five bits and the remaining three bits are used as the address of an operand. What about the accumulator's address? The Wimp51 is a class of machine called an *accumulator based* architecture and the accumulator is implied as an operand by the opcode, rather than including a register address or immediate operand. The MOV Rn,A (and five instructions) are thus two address instructions with one address mode *implied* and the other *register direct.*

Example 2. Sixteen bit addition

Assume we have two 16-bit numbers in R0,R1 and R2,R3 respectively. In other words, the first number is in the pair R0,R1 with the most significant byte (MSB) in R0 and least significant byte (LSB) in R1. We want to compute N1 = N1+N2 where N1 is the first number and N2 is the second. Start by clearing the carry bit. Since both versions of the ADDC instructions imply the accumulator as both the destination operand and one of the source operands, we will need to move the LSB of the first number to the accumulator, add it to the LSB of the other number, then move the result back to the LSB of the first number. Note that this leaves the carry out of the LSB of the sum in the carry bit, ready to add into the MSB of the 16- bit sum. We will repeat the last three instructions but use the MSB's of the two numbers instead of the LSB's. The following program is the result.

```
; compute R2,R3= R2,R3 + R0,R1
CLR C
MOV A,R1
ADDC A,R3
MOV R1,A
MOV A,R0
ADDC A,R2
MOV R0,A
```

Table 1 Wimp51 Instruction Set

Mnemonic	**Machine Code**	**Register transfer description**
MOV A,#D	01110100 dddddddd	A <= D
ADDC A,#D	00110100 dddddddd	C,A <= A+D+C
MOV Rn,A	11111nnn	Rn <= A
MOV A,Rn	11101nnn	A <= Rn
ADDC A,Rn	00111nnn	C,A<=A+Rn+C
ORL A,Rn	01001nnn	A<= A or Rn
ANL A,Rn	01011nnn	A <= A and Rn
XRL A,Rn	01101nnn	A <= A xor Rn
SWAP A	11000100	A <= A(3:0) swapped with A(7:4)
CLR C	11000011	C <= 0
SETB C	11010011	C <= 1
SJMP rel	10000000 aaaaaaaa	PC <= PC+rel+2
JZ rel	01100000 aaaaaaaa	PC <= PC+rel+2 if Z

The ORL, ANL, and XRL instructions each do bit-wise logical operations. For example, the ORL can be read as 'or logical' and results in OR-ing each bit of A with the corresponding bit of Rn and putting the result into A. Thus if R0 contains a 0x01 (abbreviated as (R0)=0x01), executing the instruction 48 results in setting the least significant bit of A and leaving the other 7 bits unchanged.

The last two instructions are both *branch* instructions. Branch instructions change the flow of control by jumping to a different location. The branch instructions on most computers uses the *program counter relative* address mode and Wimp51 is no exception. The relative address mode works by adding the immediate operand (aaaaaaaa) to the contents of the program counter (PC) to compute a new value for the program counter. This relative address, called 'rel' in the register transfer notation, is treated as a signed, 8-bit value, and is added to the incremented program counter. Thus if the instruction 80 02 is located at 0x100 and 0x101, the result is to fetch the next instruction from 0x102+02 = 0x104.

The JZ instruction is a *conditional branch* instruction. As implied by the register transfer notation, if the Z flag is set when this instruction is executed, the branch is taken, otherwise the next instruction following the JZ is executed.

Wimp51 Hardware

Figure 2 shows a block diagram of the Wimp51 CPU. The PC, ACC (or accumulator A), C and Z flags, and the eight registers R0 through R7 are the same as the programmer's model mentioned earlier. The IR is another eight bit register that is used in the control and sequencing of Wimp51. It is called the *instruction register* and holds the current instruction being executed. AUX is an eight bit temporary register that is used as the second operand for the two arithmetic and logic units, ALU and PC ALU.

The CPU has an eight bit input bus called *data*, a 16-bit output bus called *address*, an 8-bit output bus called acc_out, and a 1-bit control output called PSEN. PSEN is generated by the control unit (CU) and is active low when an instruction by is being fetched by the CPU from memory. The control unit generates several other internal control signals: ir_we, reg_we, reg_sel, reg_in, aux_we, alu_op, acc_we, pcalu_op, and pc_we. The bus *data* is latched in IR whenever ir_we is active and is latched in AUX whenever reg_in is low and aux_we is active.

The bus *acc_out* is connected to the accumulator and is latched in the register selected by reg_sel whenever reg_we is active. The register selected by reg_sel is also output and is latched by AUX whenever aux_we is active and reg_in is high. ACC latches the output of the ALU whenever acc_we is active. The C flag latches the 9th bit out of the ALU as well when acc_we is active. Z is just the NOR reduction of the ACC register as described earlier. The ALU is a combinational logic unit that computes a function of its two operands, ACC and AUX, based on the code alu_op. Alu_op is derived from the current instruction in the IR register.

The pc_alu is similar to the ALU but only needs to increment the PC or add it to AUX. Pcalu_op is a one-bit signal that controls whether pc_alu outputs PC+1 or PC+AUX.

Execution of Instructions on Wimp51

The execution of instructions on Wimp51 is similar to any other Von Neuman class machine. It starts by fetching from external memory the instruction pointed to by PC, latching the instruction in IR, decoding the instruction, fetching any operands required from the register array into AUX, and latching results (if any) in the destination register. On Wimp51 we call these steps the *fetch, decode,* and *execute* cycles. Figure 3 shows a state diagram of the Wimp51 control unit.

Each state takes exactly one clock cycle. Each instruction thus takes three clock cycles. The sequence starts by asserting PSEN and ir_we during the fetch state. This causes external memory to respond by driving the data bus with the contents of memory at location (PC). At the end of the fetch state, PSEN and ir_we are de-asserted and the value on the data bus is latched in IR. PC is also incremented. During the decode state, reg_sel is driven by the three least significant bits of IR, and reg_in is driven high by the CU. If a register operand is actually required, aux_we is asserted. If the immediate address mode is active, then PSEN is again asserted and reg_in is driven low, but this time the data is latched in AUX. The micro-operations during the execute state depend on whether a branch instruction is in IR or not. If a branch instruction is active, PC is possibly loaded with the sum of PC and AUX. If IR contains an instruction other than a branch, ACC or a register is possibly written with a result.

At least one byte is thus fetched into IR during the fetch cycle. A second byte is fetched into AUX during the decode cycle if the instruction is a branch or immediate mode instruction. Results are written to the destination register at the end of the execute state.

Summary

This section has shown:

- The Wimp51 programmer's model
- The instruction set of the Wimp51
- The hardware block diagram of the Wimp51
- The sequence of operations that take place during execution of Wimp51 instructions.

Questions

Answer the following questions.

1. The Wimp51 does not have a subtract instruction. Write a sequence of Wimp51 instructions that will compute R0 = R1 – R2.
2. Write a sequence of Wimp51 instructions that will increment (add 1 to) R0.
3. Write a sequence of instructions that uses ANL to clear bit 5 of R3.
4. Write a sequence of instructions that uses XRL to complement bit 2 of R4.
5. How many functions must the ALU implement in order to execute all of the Wimp51 instructions? Describe the functions that ALU must implement.
6. A typical *bit-slice* of the ALU will have n+3 inputs and two outputs: n alu_op code bits, the two operand bits, a carry in bit, a carry out bit, and the ALU out bit. Make a suitable assignment for the n code bits and implement the ALU bit slice using AND, OR, and NOT gates. You should be able to cascade eight of your bit slices to get a complete ALU.
7. Use the WWW (or other reference source) to find out when Von Neuman's group developed the stored program concept, and the name of the machine the group was building when the concept was developed.

Figures

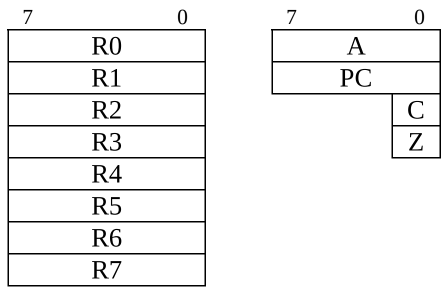

Figure 1 Programmer's model

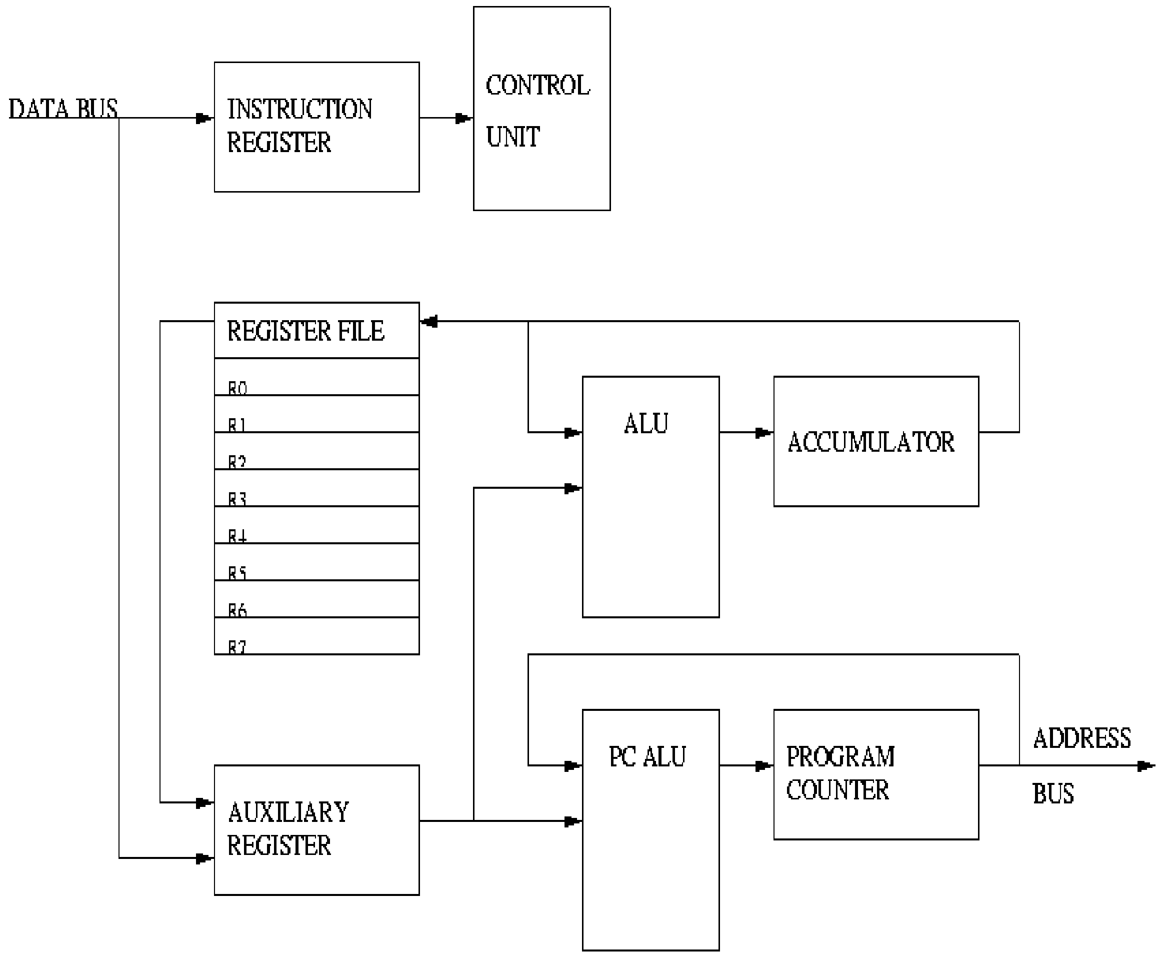

Figure 2 Wimp51 Block Diagram