Some important questions

|  |
| --- |
| What is the instruction cycle? Explain with the help of flow chart. |
| What do you mean by Integrated circuits? |
| Explain the instruction cycle with the help of a flowchart. |
| Write any three logic micro operations. |
| Write any three shift micro operations. |
| Explain the use of one address instructions. |
| Explain selective set and mask micro operations. |
| what do you understand by computer organization, design and architecture? |
| What is register transfer language (RTL)? |
|  |
| Explain the various modes of transfer? |

**5.1. Instruction Codes**

**5.1.1. Computer Instructions**

Computer instructions are the basic components of a machine language program. They are also known as *macrooperations*, since each one is comprised of a sequences of microoperations.

Each instruction initiates a sequence of microoperations that fetch operands from registers or memory, possibly perform arithmetic, logic, or shift operations, and store results in registers or memory.

Instructions are encoded as binary *instruction codes*. Each instruction code contains of a *operation code*, or *opcode*, which designates the overall purpose of the instruction (e.g. add, subtract, move, input, etc.). The number of bits allocated for the opcode determined how many different instructions the architecture supports.

In addition to the opcode, many instructions also contain one or more *operands*, which indicate where in registers or memory the data required for the operation is located. For example, and add instruction requires two operands, and a not instruction requires one.

15 12 11 6 5 0

+-----------------------------------+

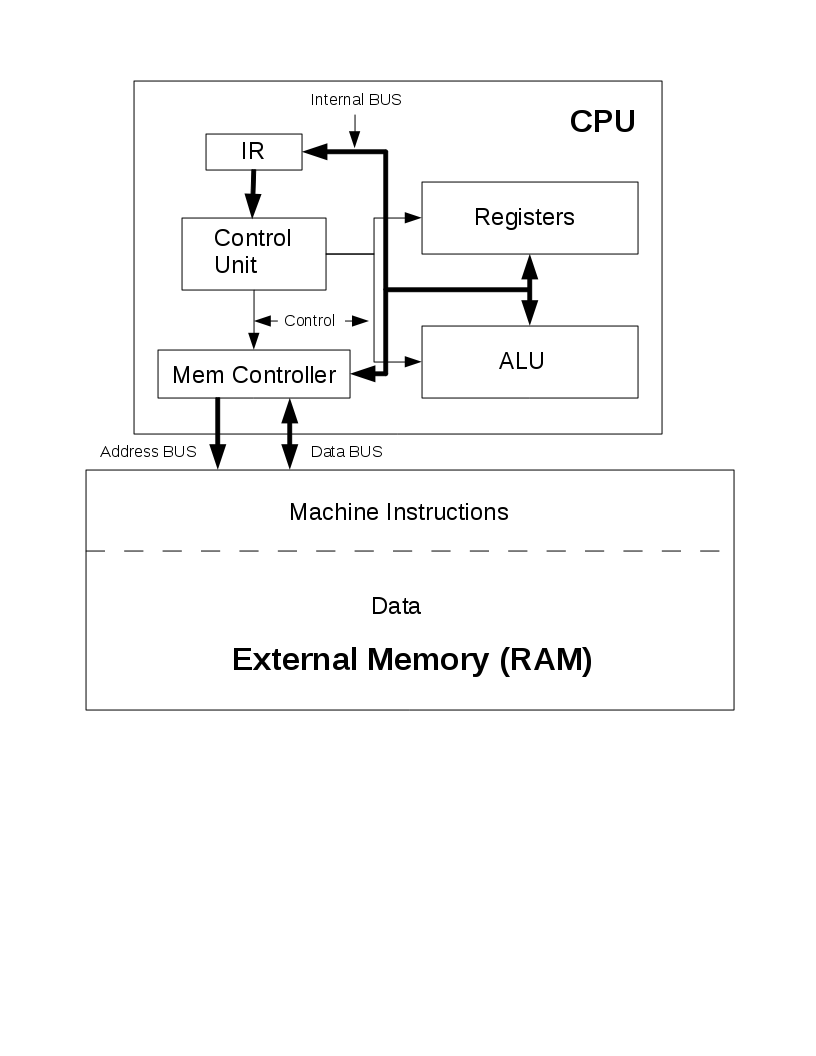
| Opcode | Operand | Operand |

+-----------------------------------+

The opcode and operands are most often encoded as unsigned binary numbers in order to minimize the number of bits used to store them. For example, a 4-bit opcode encoded as a binary number could represent up to 16 different operations.

The *control unit* is responsible for decoding the opcode and operand bits in the instruction register, and then generating the control signals necessary to drive all other hardware in the CPU to perform the sequence of microoperations that comprise the instruction.

**Figure 5.1. CPU Block Diagram**



**5.1.2. Stored Program Organization**

**5.1.2.1. Von Neumann and Harvard Architectures**

In the Von Neumann architecture, machine instructions and data are stored in the same RAM during program execution.

Book figure 5-1:

A Harvard architecture CPU, in contrast, stored the program and the data in separate memory units. For example, many microcontrollers store the program in Flash memory and the data in traditional, volatile RAM.

**5.1.2.2. Operand-based Architecture Classification**

Architectures are also classified according to how instructions access memory and process data:

* Memory-to-memory: Most instructions can access memory for any operand. The VAX architecture from Digital Equipment Corporation is an example.
* addl3 x, y, sum # x, y, and sum are memory addresses

* Register-memory: Instructions allow only one operand to be a memory address, while the other(s) must be CPU registers. The x86 architecture is an example.
* movl eax, x
* addl eax, y
* movl sum, eax

* Load-store: Only load and store instructions can access memory. All others must use CPU registers for all operands. The MIPS processor, originally from Digital Equipment Corporation is an example.
* lw $t0, x
* lw $t1, y
* add $t0, $t0, $t1
* sw $t0, sum

* Accumulator-based: One special register, called the accumulator (AC), is an implied operand for most operations. The Zylog Z80 is an example.
* load x # AC ← x
* add y # AC ← AC + y
* store sum # sum ←- AC

**5.1.2.3. Designing an Instruction Code**

Machine instruction codes may all be the same length (e.g. MIPS processor), or codes for different instructions may be different lengths (e.g. x86, VAX processors).

Suppose all instruction codes of a hypothetical accumulator-based CPU are exactly 16 bits. A simple instruction code format could consist of a 4-bit operation code (opcode) and a 12-bit memory address.

15 12 11 0

+-----------------------+

| Opcode | Address |

+-----------------------+

This would allow for how many different instructions? How much memory?

Suppose a program contains two variables, x and y. The variable x represents address 0x010 and y represents address 0x012. A segment of the *list file*, which shows machine code and assembly code side-by-side, might appear as follows:

0 010 add x

1 012 sub y

We see that the opcode for add is 000 (0x0) and the opcode for sub is 001 (0x1).

The format above represents *memory-reference* instructions, which act on an operand from memory and the accumulator. Not all operations require a second operand, so some instructions could act on the accumulator alone. In this case, address bits can be used for other purposes. For example:

clr # AC = 0

neg # AC = -AC

not # AC = AC'

inc # AC = AC + 1

One or more patterns in the 4-bit opcode can be used to signify that the other 12 bits specify an operation instead of an address. This reduces the number of memory-reference instructions possible, but increases the overall instruction count.

0XXX add XXX

1XXX sub XXX

...

F000 clr

F001 neg

F002 not

F003 inc

How many memory-reference instructions can this CPU have?

How many non-memory-reference instructions can this CPU have?

As a second example, suppose a load-store architecture computer has 2-operand instructions, 32 registers, 1 megabyte of byte-addressable memory, 4 addressing modes, 50 register-reference instructions, and 6 load-store instructions. What would the instruction code look like for register-reference instructions? What would the instruction code look like for memory-reference instructions?

Solution: Since there are 50 register-reference instructions, we will need 6 bits for the opcode. (6 bits allows for up to 26 unique opcodes.) With 32 registers, we will need 5 bits to specify each register, so the instruction code format will be 16 bits:

+--------------------------+

| opcode | reg1 | reg2 |

+--------------------------+

6 5 5

For 6 load-store opcodes, we need 3 bits. 4 addressing modes requires 2 bits to go with the address. A possible instruction code format is as follows:

+-------------------------------+

| opcode | reg | mode | address |

+-------------------------------+

3 5 4 20

As a third example, suppose a register-memory architecture has 8 registers, a 64k memory capacity, 100 instructions, and 6 addressing modes. Design an instruction code format for memory-reference instructions.

Solution: To represent 100 instructions, we will need 7 bits for the opcode. We'll need 16 bits for a memory address for 64k memory, 3 bits to represent one of 8 registers, and 3 bits to cover all 6 addressing modes for the memory operand. One possible instruction format is shown below. Since this adds up to 19 bits, we would likely use 24 bits for the instruction code to make it fit well into byte-addressable memory. The additional bits could be used to support more opcodes and/or addressing modes.

+-------------------------------+

| opcode | reg | mode | address |

+-------------------------------+

7 3 3 16

Suppose a direct address in assembly language is represented by a label, as in x below, and an indirect address by a label in parentheses, as in (ptr) below.

mov x, r3

add (ptr), r3

If the opcode for mov is 00000001, add is 0000010, the mode bits for direct addressing are 100, and the bits for indirect addressing are 101, the address x is 0x00F0, and ptr is 0x00F2, the instruction codes for the two instructions above would be:

0000001 011 100 0000000011110000

0000010 011 101 0000000011110010

Design an instruction code format for a memory-to-memory architecture with 16 registers, a 4 gigabyte memory capacity, 250 instructions, and 16 addressing modes. Assume that there are anywhere from 0 to 3 operands per instruction, and each operand may be in a register or in memory.

**5.1.2.4. Some Common Addressing Modes**

* Direct: Instruction code contains address of operand
* 0 005 AC = AC + contents of address 5

\* 1 memory-reference beyond fetching instruction

* Immediate: Instruction code contains operand
* 1 005 AC = AC + 5

\* No memory-reference after fetching instruction

* Indirect: Instruction code contains address of address of operand
* 2 005 AC = AC + contents of address stored at address 5

\* 2 memory-references after fetching instruction

Effective address = actual address of the data in memory.

**Table 5.1. Effective Address by Addressing Mode**

| **Mode** | **Effective Address** |
| --- | --- |
| Immediate | Address of the instruction itself |
| Direct | Address contained in the instruction code |
| Indirect | Address at the address in the instruction code |

**5.1.2.5. Basic Computer Instruction Format**

The Basic Computer has a 16-bit instruction code similar to the examples described above. It supports direct and indirect addressing modes.

How many bits are required to specify the addressing mode?

15 14 12 11 0

+------------------+

| I | OP | ADDRESS |

+------------------+

I = 0: direct

I = 1: indirect

## 5.3. Computer Instructions

All Basic Computer instruction codes are 16 bits wide. There are 3 instruction code formats:

* Memory-reference instructions take a single memory address as an operand, and have the format:
* 15 14 12 11 0
* +-------------------+
* | I | OP | Address |
* +-------------------+

If I = 0, the instruction uses direct addressing. If I = 1, addressing in indirect.

How many memory-reference instructions can exist?

* Register-reference instructions operate solely on the AC register, and have the following format:
* 15 14 12 11 0
* +------------------+
* | 0 | 111 | OP |
* +------------------+

How many register-reference instructions can exist? How many memory-reference instructions can coexist with register-reference instructions?

* Input/output instructions have the following format:
* 15 14 12 11 0
* +------------------+
* | 1 | 111 | OP |
* +------------------+

**5.4. Timing and Control**

All sequential circuits in the Basic Computer CPU are driven by a master clock, with the exception of the INPR register.

At each clock pulse, the control unit sends control signals to control inputs of the bus, the registers, and the ALU.

Control unit design and implementation can be done by two general methods:

* A *hardwired* control unit is designed from scratch using traditional digital logic design techniques to produce a minimal, optimized circuit. In other words, the control unit is like an ASIC (application-specific integrated circuit).
* A *microprogrammed* control unit is built from some sort of ROM. The desired control signals are simply stored in the ROM, and retrieved in sequence to drive the microoperations needed by a particular instruction.

In this chapter, we are designing a hardwired control unit.

What are the advantages of each type of control unit design?

Figure 5-6 shows how some of the control signals are generated using the instruction code and sequence counter.

From here on, we use primarily the timing signals and bits from the instruction register to construct control functions. These are the independent variables in the CPU. Control inputs on CPU components, such as ACload, S0, etc. are dependent variables, i.e. functions of the timing signals and IR bits.

The following components provide most of the independent variables for the Boolean functions driving control inputs such as ACLD, PCINR, etc.

* I flip-flop
* Opcode field from the instruction code
* Address bits from the instruction code
* 4-bit counter with INR, CLR (CLR=1 at power-on)

The sequence counter along with the decoder on its outputs generate a regular sequence of timing signals that we will refer to as T0, T1, etc.

Figure 5-7 shows how timing signals are generated from the sequence counter and a decoder, and how they interact with another control variable D3.

**5.5. Instruction Cycle**

In this chapter, we examine the sequences of microoperations that the Basic Computer goes through for each instruction. Here, you should begin to understand how the required control signals for each state of the CPU are determined, and how they are generated by the control unit.

The CPU performs a sequence of microoperations for each instruction. The sequence for each instruction of the Basic Computer can be refined into 4 abstract phases:

1. Fetch instruction
2. Decode
3. Fetch operand
4. Execute

Program execution can be represented as a top-down design:

1. Program execution
   1. Instruction 1
      1. Fetch instruction
      2. Decode
      3. Fetch operand
      4. Execute
   2. Instruction 2
      1. Fetch instruction
      2. Decode
      3. Fetch operand
      4. Execute
   3. Instruction 3 ...

Program execution begins with:

PC ← address of first instruction, SC ← 0

After this, the SC is incremented at each clock cycle until an instruction is completed, and then it is cleared to begin the next instruction. This process repeats until a HLT instruction is executed, or until the power is shut off.

**5.5.1. Instruction Fetch and Decode**

The instruction fetch and decode phases are the same for all instructions, so the control functions and microoperations will be independent of the instruction code.

Everything that happens in this phase is driven entirely by timing variables T0, T1 and T2. Hence, all control inputs in the CPU during fetch and decode are functions of these three variables alone.

T0: AR ← PC

T1: IR ← M[AR], PC ← PC + 1

T2: D0-7 ← decoded IR(12-14), AR ← IR(0-11), I ← IR(15)

For every timing cycle, we assume SC ← SC + 1 unless it is stated that SC ← 0.

The operation D0-7 ← decoded IR(12-14) is not a register transfer like most of our microoperations, but is actually an inevitable consequence of loading a value into the IR register. Since the IR outputs 12-14 are directly connected to a decoder, the outputs of that decoder will change as soon as the new values of IR(12-14) propagate through the decoder.

Note that incrementing the PC at time T1 assumes that the next instruction is at the next address. This may not be the case if the current instruction is a branch instruction. However, performing the increment here will save time if the next instruction immediately follows, and will do no harm if it doesn't. The incremented PC value is simply overwritten by branch instructions.

In hardware development, unlike serial software development, it is often advantageous to perform work that may not be necessary. Since we can perform multiple microoperations at the same time, we might was well do everything that *might* be useful at the earliest possible time.

Likewise, loading AR with the address field from IR at T2 is only useful if the instruction is a memory-reference instruction. We won't know this until T3, but there is no reason to wait since there is no harm in loading AR immediately.

Figure 5-8 shows part of the implementation of the CPU for the first two clock pulses of the instruction cycle. Note that each control input for the bus, registers, and ALU is a Boolean function with multiple terms. Hence, the OR gates in the diagram.

*The goal in control unit design is to determine the Boolean function needed for each control input of the registers, bus, and ALU.*

A few things to note:

* T0 implies S2'S1S0' and ARLD
* What does T1 imply?

Each control input is simply a Boolean function of timing variables and other variables, mostly originating from the instruction code. This is the basis of the entire control unit design!

Each state of the CPU provides one term for some of the control input functions:

* S0 = T1 + ...
* S1 = T0 + T1 + ...
* S2 = ?
* ARLD = ?

**5.5.2. Determining the Instruction Type**

Figure 5-9 shows the process of determining the type of instruction. Refer to figure 5-5 for a a reminder of the instruction code formats.

By time T2, the opcode has been decoded by the decoder attached to IR(12-14), and the control signals D0-7 are available. At pulse T2, IR(15) is loaded into the I flip-flop. Hence, all of these signals are available for use at pulse T3.

D7 indicates that the opcode field is 111, and this is either a register or I/O instruction. (i.e. it is not a memory-reference instruction.)

The I bit allows us to distinguish between register and I/O instructions.

D7' indicates a memory-reference instruction. In this case, the I bit determines the addressing mode. What happens at time T3 therefore depends on the two variables D7 and I.

* Register-reference:

D7I'T3: Execute register-reference instruction.

* I/O:

D7IT3: Execute I/O instruction.

* Memory-reference with indirect addressing:

D7'IT3: AR ← M[AR]

* Memory-reference with direct addressing:

D7'I'T3: Nothing. Effective address is already in AR. This wastes a clock cycle when direct addressing is used, but it simplifies the memory-reference execute phase by ensuring tha the CPU is in a known state at time T4.

**5.5.3. Register-reference Execute Phase**

The control function D7I' indicates a register-reference instruction, but which one?

Regardless of which instruction it is, T3 will be the last timing cycle for this instruction, and we will want the next clock pulse to start the timing cycle over at T0 and fetch the next instruction.

Since D7I'T3 is common to all register-reference instructions, we will abbreviate it as simple 'r'.

D7I'T3: SC ← 0

r: SC ← 0

What is the Boolean function for SCCLR?

The CLA instruction is indicated by instruction code 780016 (0 111 1000000000002).

The leftmost 4 bits indicate only that this is a register-reference instruction. The rightmost 12 bits indicate that it is the CLA instruction.

How many register-reference instructions are possible with 12 bits to represent the opcode?

The Basic Computer does not encode the bits for register-reference instructions as a binary number, but instead uses them directly. Hence, only one of these bits can be 1 for a given instruction, and we are limited to 12 register-reference instructions.

For the register-reference execute phase, all control inputs in the CPU are functions of T3, r and one of the variables B0 through B11, which come directly from IR(0-11).

CLA 10000000

CLE 01000000

CMA 00100000

...

HLT 00000001

Can we create register-reference instructions that combine two or more operations, e.g. CLACLE, CLACMA, CLASZA?

Since B11 indicates a CLA instruction the execute cycle for CLA is driven by the function rB11.

rB11: AC ← 0

ACCLR = ?

The "skip" instructions, SPA, SNA, SZA, and SZE are primitive branch instructions that simply skip the next instruction if the condition is met. For example, SPA (skip on positive accumulator) performs the following:

if ( AC >= 0 ) then PC ← PC + 1

**Note**

Technically, the term *positive* means strictly greater than zero. The SPA instruction actually branches if AC is *non-negative*.

Skipping the next instruction is a common occurrence in machine code, even where more sophisticated branch instructions are available. We often see a conditional branch followed by an unconditional branch. Having conditional skip and unconditional branch instructions is sufficient to implement any program.

How do we determine when AC >= 0? AC < 0? AC = 0? AC > 0?

What are the complete control functions and microoperations for the execute cycle of each of the following? ( Table 5-1 is not complete. )

* SPA
* SNA
* SZA
* SZE

Show the connections from the AC register to the PC INR input.

**5.5. Instruction Cycle**

In this chapter, we examine the sequences of microoperations that the Basic Computer goes through for each instruction. Here, you should begin to understand how the required control signals for each state of the CPU are determined, and how they are generated by the control unit.

The CPU performs a sequence of microoperations for each instruction. The sequence for each instruction of the Basic Computer can be refined into 4 abstract phases:

1. Fetch instruction
2. Decode
3. Fetch operand
4. Execute

Program execution can be represented as a top-down design:

1. Program execution
   1. Instruction 1
      1. Fetch instruction
      2. Decode
      3. Fetch operand
      4. Execute
   2. Instruction 2
      1. Fetch instruction
      2. Decode
      3. Fetch operand
      4. Execute
   3. Instruction 3 ...

Program execution begins with:

PC ← address of first instruction, SC ← 0

After this, the SC is incremented at each clock cycle until an instruction is completed, and then it is cleared to begin the next instruction. This process repeats until a HLT instruction is executed, or until the power is shut off.

**5.5.1. Instruction Fetch and Decode**

The instruction fetch and decode phases are the same for all instructions, so the control functions and microoperations will be independent of the instruction code.

Everything that happens in this phase is driven entirely by timing variables T0, T1 and T2. Hence, all control inputs in the CPU during fetch and decode are functions of these three variables alone.

T0: AR ← PC

T1: IR ← M[AR], PC ← PC + 1

T2: D0-7 ← decoded IR(12-14), AR ← IR(0-11), I ← IR(15)

For every timing cycle, we assume SC ← SC + 1 unless it is stated that SC ← 0.

The operation D0-7 ← decoded IR(12-14) is not a register transfer like most of our microoperations, but is actually an inevitable consequence of loading a value into the IR register. Since the IR outputs 12-14 are directly connected to a decoder, the outputs of that decoder will change as soon as the new values of IR(12-14) propagate through the decoder.

Note that incrementing the PC at time T1 assumes that the next instruction is at the next address. This may not be the case if the current instruction is a branch instruction. However, performing the increment here will save time if the next instruction immediately follows, and will do no harm if it doesn't. The incremented PC value is simply overwritten by branch instructions.

In hardware development, unlike serial software development, it is often advantageous to perform work that may not be necessary. Since we can perform multiple microoperations at the same time, we might was well do everything that *might* be useful at the earliest possible time.

Likewise, loading AR with the address field from IR at T2 is only useful if the instruction is a memory-reference instruction. We won't know this until T3, but there is no reason to wait since there is no harm in loading AR immediately.

Figure 5-8 shows part of the implementation of the CPU for the first two clock pulses of the instruction cycle. Note that each control input for the bus, registers, and ALU is a Boolean function with multiple terms. Hence, the OR gates in the diagram.

*The goal in control unit design is to determine the Boolean function needed for each control input of the registers, bus, and ALU.*

A few things to note:

* T0 implies S2'S1S0' and ARLD
* What does T1 imply?

Each control input is simply a Boolean function of timing variables and other variables, mostly originating from the instruction code. This is the basis of the entire control unit design!

Each state of the CPU provides one term for some of the control input functions:

* S0 = T1 + ...
* S1 = T0 + T1 + ...
* S2 = ?
* ARLD = ?

**5.5.2. Determining the Instruction Type**

Figure 5-9 shows the process of determining the type of instruction. Refer to figure 5-5 for a a reminder of the instruction code formats.

By time T2, the opcode has been decoded by the decoder attached to IR(12-14), and the control signals D0-7 are available. At pulse T2, IR(15) is loaded into the I flip-flop. Hence, all of these signals are available for use at pulse T3.

D7 indicates that the opcode field is 111, and this is either a register or I/O instruction. (i.e. it is not a memory-reference instruction.)

The I bit allows us to distinguish between register and I/O instructions.

D7' indicates a memory-reference instruction. In this case, the I bit determines the addressing mode. What happens at time T3 therefore depends on the two variables D7 and I.

* Register-reference:

D7I'T3: Execute register-reference instruction.

* I/O:

D7IT3: Execute I/O instruction.

* Memory-reference with indirect addressing:

D7'IT3: AR ← M[AR]

* Memory-reference with direct addressing:

D7'I'T3: Nothing. Effective address is already in AR. This wastes a clock cycle when direct addressing is used, but it simplifies the memory-reference execute phase by ensuring tha the CPU is in a known state at time T4.

**5.5.3. Register-reference Execute Phase**

The control function D7I' indicates a register-reference instruction, but which one?

Regardless of which instruction it is, T3 will be the last timing cycle for this instruction, and we will want the next clock pulse to start the timing cycle over at T0 and fetch the next instruction.

Since D7I'T3 is common to all register-reference instructions, we will abbreviate it as simple 'r'.

D7I'T3: SC ← 0

r: SC ← 0

What is the Boolean function for SCCLR?

The CLA instruction is indicated by instruction code 780016 (0 111 1000000000002).

The leftmost 4 bits indicate only that this is a register-reference instruction. The rightmost 12 bits indicate that it is the CLA instruction.

How many register-reference instructions are possible with 12 bits to represent the opcode?

The Basic Computer does not encode the bits for register-reference instructions as a binary number, but instead uses them directly. Hence, only one of these bits can be 1 for a given instruction, and we are limited to 12 register-reference instructions.

For the register-reference execute phase, all control inputs in the CPU are functions of T3, r and one of the variables B0 through B11, which come directly from IR(0-11).

CLA 10000000

CLE 01000000

CMA 00100000

...

HLT 00000001

Can we create register-reference instructions that combine two or more operations, e.g. CLACLE, CLACMA, CLASZA?

Since B11 indicates a CLA instruction the execute cycle for CLA is driven by the function rB11.

rB11: AC ← 0

ACCLR = ?

The "skip" instructions, SPA, SNA, SZA, and SZE are primitive branch instructions that simply skip the next instruction if the condition is met. For example, SPA (skip on positive accumulator) performs the following:

if ( AC >= 0 ) then PC ← PC + 1

**Note**

Technically, the term *positive* means strictly greater than zero. The SPA instruction actually branches if AC is *non-negative*.

Skipping the next instruction is a common occurrence in machine code, even where more sophisticated branch instructions are available. We often see a conditional branch followed by an unconditional branch. Having conditional skip and unconditional branch instructions is sufficient to implement any program.

How do we determine when AC >= 0? AC < 0? AC = 0? AC > 0?

What are the complete control functions and microoperations for the execute cycle of each of the following? ( Table 5-1 is not complete. )

* SPA
* SNA
* SZA
* SZE

Show the connections from the AC register to the PC INR input.

**5.7. Input-Output and Interrupt**

**5.7.1. Hardware Summary**

The Basic Computer I/O consists of a simple terminal with a keyboard and a printer/monitor.

The keyboard is connected serially (1 data wire) to the INPR register. INPR is a shift register capable of shifting in external data from the keyboard one bit at a time. INPR outputs are connected in parallel to the ALU.

Shift enable

|

v

+-----------+ 1 +-------+

| Keyboard |---/-->| INPR <|--- serial I/O clock

+-----------+ +-------+

|

/ 8

| | |

v v v

+---------------+

| ALU |

+---------------+

|

/ 16

|

v

+---------------+

| AC <|--- CPU master clock

+---------------+

How many CPU clock cycles are needed to transfer a character from the keyboard to the INPR register? (tricky)

Are the clock pulses provided by the CPU master clock?

RS232, USB, Firewire are serial interfaces with their own clock independent of the CPU. ( USB speed is independent of processor speed. )

* RS232: 115,200 kbps (some faster)
* USB: 11 mbps
* USB2: 480 mbps
* FW400: 400 mbps
* FW800: 800 mbps
* USB3: 4.8 gbps

OUTR inputs are connected to the bus in parallel, and the output is connected serially to the terminal. OUTR is another shift register, and the printer/monitor receives an end-bit during each clock pulse.

**5.7.2. I/O Operations**

Since input and output devices are not under the full control of the CPU (I/O events are asynchronous), the CPU must somehow be told when an input device has new input ready to send, and an output device is ready to receive more output.

The FGI flip-flop is set to 1 after a new character is shifted into INPR. This is done by the I/O interface, not by the control unit. This is an example of an asynchronous input event (not synchronized with or controlled by the CPU).

The FGI flip-flop must be cleared after transferring the INPR to AC. This must be done as a microoperation controlled by the CU, so we must include it in the CU design.

The FGO flip-flop is set to 1 by the I/O interface after the terminal has finished displaying the last character sent. It must be cleared by the CPU after transferring a character into OUTR.

Since the keyboard controller only sets FGI and the CPU only clears it, a JK flip-flop is convenient:

+-------+

Keyboard controller --->| J Q |----->

| | |

+--------\-----\ | |

) or >----->|> FGI |

+--------/-----/ | |

| | |

CPU-------------------->| K |

+-------+

How do we control the CK input on the FGI flip-flop? (Assume leading-edge triggering.)

There are two common methods for detecting when I/O devices are ready, namely *software polling* and *interrupts*. These two methods are discussed in the following sections.

Table 5-5 outlines the Basic Computer input-output instructions.

**5.7.3. Software Polling**

In software polling, the software is responsible for checking the status of I/O devices and initiating transactions when the device is ready. The simplest form of software polling is called *spin waiting*. A *spin waiting* loop does nothing but watch the status of a device until it becomes ready. When it is ready, the loop exits and the I/O transaction is performed.

key\_wait, ski

bun key\_wait

inp

/ Do something with the input

What are the pros and cons of spin-waiting?

A fast typist can do perhaps 10 characters per second at best. If the instruction cycle takes 1 microsecond (a very slow CPU), the spin wait loop can check the keyboard status 500,000 times per second, or 50,000 times between keystrokes.

Rather than waste 50,000 instruction cycles between keystrokes, we could put these instruction cycles to use by adding some computational code to the loop, and poll periodically throughout the code.

loop, / compute something

/ If input available, go to get\_input

ski

bun no\_input1

bun get\_input

no\_input1, / compute something else

/ If input available, go to get\_input

ski

bun loop

get\_input, inp

/ Do something with the input

bun loop

The problem with periodic polling is that it is difficult to ensure polling at regular intervals, since computational code contains conditional statements, and follows different paths depending on the input.

What happens when we poll too frequently?

What happens when we don't poll frequently enough?

Ellipse function, serial terminal.

**5.7.4. Interrupts**

To alleviate the problems of software polling, a hardware solution is needed.

Analogies to software polling in daily life tend to look rather silly. For example, imagine a teacher is analogous to a CPU, and the students are I/O devices. The students are working asynchronously, as the teacher walks around the room constantly asking each individual student "are you done yet?".

What would be a better approach?

With interrupts, the running program is not responsible for checking the status of I/O devices. Instead, it simply does its own work, and assumes that I/O will take care of itself!

When a device becomes ready, the CPU *hardware* initiates a branch to an I/O subprogram called an *interrupt service routine (ISR)*, which handles the I/O transaction with the device.

An interrupt can occur during *any* instruction cycle as long as interrupts are enabled. When the current instruction completes, the CPU interrupts the flow of the program, executes the ISR, and then resumes the program. The program itself is not involved and is in fact unaware that it has been interrupted.

Figure 5-13 outlines the Basic Computer interrupt process.

Interrupts can be globally enabled or disabled via the IEN flag (flip-flop).

Some architectures have a separate ISR for each device. The Basic Computer has a single ISR that services both the input and output devices.

If interrupts are enabled, then when either FGI or FGO gets set, the R flag also gets set. (R = FGI v FGO) This allows the system to easily check whether *any* I/O device needs service. Determining which one needs service can be done by the ISR.

If R = 0, the CPU goes through a normal instruction cycle. If R = 1, the CPU branches to the ISR to process an I/O transaction.

How much time does checking for interrupts add to the instruction cycle?

Interrupts are usually disabled while the ISR is running, since it is difficult to make an ISR *reentrant*. (Callable while it is already in progress, such as a recursive function.) Hence, IEN and R are cleared as part of the interrupt cycle. IEN should be re-enabled by the ISR when it is finished. ( In many architectures this is done by a special return instruction to ensure that interrupts are not enabled before the return is actually executed. )

The Basic Computer interrupt cycle is shown in figure 5-13 (above).

Figure 5-14 shows a memory map including the ISR and user program.

The Basic Computer interrupt cycle in detail:

T0'T1'T2'(IEN)(FGI v FGO): R ← 1

RT0: AR ← 0, TR ← PC

RT1: M[AR] ← TR, PC ← 0

RT2: PC ← PC + 1, IEN ← 0, R ← 0, SC ← 0

To enable the use of interrupts requires several steps:

1. Write an ISR
2. Install the ISR in memory at some arbitrary address X
3. Install the instruction "BUN X" at address 1
4. Enable interrupts with the ION instruction

The sequence of events utilizing an interrupt to process keyboard input is as follows:

1. A character is typed
2. FGI ← 1 (same as with polling)
3. R ← 1, IEN ← 0
4. M[0] ← PC (store return address)
5. PC ← 1 (branch to interrupt vector)
6. BUN X (branch to ISR)
7. ISR checks FGI (found to be 1)
8. INP (AC ← INPR)
9. Character in AC is placed in a queue
10. ISR checks FGO (found to be 0)
11. ION
12. BUN 0 I

Programs then read their input from a queue rather than directly from the input device. The ISR adds input to the queue as soon as it is typed, regardless of what code is running, and then returns to the running program.