7 BRANCHING AND TESTING

As mentioned earlier, for an instruction set to be "general purpose" it must include four types of instructions: data transfer, arithmetic/logic, testing, and branching. The previous examples of instruction sets have included only data transfer and arithmetic/logic instructions. In what follows, the remaining types of instructions, testing and branching, will be described.

7.1 Testing

Testing requires the evaluation of a conditional expression; that is, an expression that evaluates to "true" or "false." For the CPU to make use of a test, the result of the test must be stored. Therefore CPU designs usually employ a special purpose register, called a "condition code register" for that purpose.

In hardware design, a register is constructed from a set of 1-bit storage devices called "flip-flops." Flip-flops are therefore 1-bit registers. Different architectures may implement a condition code register for storing the results of k different tests with s a single register of length k or with k individual flip-flops. When implemented with a single register, each bit position has a designated role. When a set of flip-flops is used, each flip-flop is responsible for recording the results of a specific test.

In the following examples of test instructions, it is assumed that a single register, called the "CCR", will be used and consists of 4 bits whose roles are as follows:

bit 0 indicates the result of a test for equality. The value is stored in CCR(0).

bit 1 indicates the result of a test for "less than". The bit value is stored in CCR(1).

bit 2 indicates the value of the carry-out in unsigned addition. The bit value is stored in CCR(2).

bit 3 indicates whether a signed arithmetic operation results in an overflow or underflow. The value is stored in CCR(3).

7.2 Types of Tests

The purpose of a test can be classified. Most assembly language instruction sets provide test instructions to accommodate all of the following types:

1. **Comparison of Two Values**: The typical arithmetic comparison of two values A and B provides for three possible outcomes: $A < B$, $A = B$, or $A > B$. Like the usual arithmetic operations, $<$, $=$, and $>$ are operators with two source operands, A and B, and a destination operand, which is a boolean value indicating the result of the test. Therefore, test instructions have formats similar to arithmetic instructions.

Some examples are as follows:

(a) **direct mode**:

INSTRUCTION		accesses	
Syntax	Semantics	fetch	exec
TSTM Rj, A	CCR(0) ← M[A] = R[j] CCR(1) ← M[A] < R[ij	2	1

(b) **register mode**:

INSTRUCTION		accesses	
Syntax	Semantics	fetch	exec
TSTR Ri, Rj	CCR(0) ← R[j] = R[i] CCR(1) ← R[j] < R[i]	1	0

2. **Sign Tests**: Sign tests are really a special case of comparison tests, since one of the operands is typically 0. Test instructions to implement this special case are useful because they permit one operand to be treated as implicit. For example:

INSTRUCTION		accesses	
Syntax	Semantics	fetch	exec
TST0 Rj	CCR(0) ← R[j] = 0 CCR(1) ← R[j] < 0	1	0

3. **Validity Tests**: In order to write robust programs that detect when computations are invalid, test instructions can include checking for overflow and underflow. In the example CCR provided, CCR(2) indicates when overflow has occurred as a result of unsigned arithmetic, while CCR(3) indicates when overflow or underflow has occurred as a result of signed arithmetic

4. **External Requests** : The CPU can communicate with the other components of a computer via designated control signal lines. One particularly important such signal line is an "interrupt."

   An **interrupt** is a signal generated externally by the hardware or the operating system, that can be used to control the behaviour of a program.

   Instructions capable of checking for interrupts make it possible for computers to assign tasks to other components, and then continue executing other instructions, while checking periodically for the status of the other components. In turn, these components can gain control of the CPU by signalling a request for service using an interrupt signal

7.3 Branching

A branching instruction is one that can change where the CPU retrieves the next instruction in the instruction execution algorithm. Most CPU's keep track of the location of the next instruction with the aid of another register called the "program counter (PC)." When a program is assembled, the loader initializes the PC with the starting address of the program; that is, the first executable instruction of the program.

The instruction execution algorithm then get the instruction located at the address specified by the PC. For the program to proceed sequentially through a sequence of instructions, the PC must be changed to the address of the next instruction. This change can occur by incrementing the address stored in the PC provided the next instruction to be executed is located in memory in the next location after the instruction currently being executed.

However it may be necessary, as the result of a test, to skip the next instructions that follow sequentially in memory, and continue execution at some other location. Branch instructions allow the programmer to change the contents of the PC to hold a specific address. The value to be stored in the PC is called the "branch address."

Some example branch instructions are as follows:

1. **immediate mode**: The branch address is provided explicitly as an operand.

INSTRUCTION		accesses	
Syntax	Semantics	fetch	exec
JUMP addr	PC ← addr	1	0

2. **register indirect mode**: The branch address is stored in a location of internal memory (i.e., a register)

INSTRUCTION		accesses	
Syntax	Semantics	fetch	exec
JUMP Rj	PC ← R[j]	1	0

3. **relative mode**: The branch address is calculated using the address of the instruction currently being executed and the distance the branch address is from the location of the current instruction. The distance requires fewer bits than the actual address to encode in the instruction and so a shorter instruction is obtained.

INSTRUCTION		accesses	
Syntax	Semantics	fetch	exec
JMPLT addr	IF CCR(0) THEN PC ← PC + addr - (* + 1)	1	0

NOTE: Here the address of the current instruction is denoted by " * ".

8 x86-64 INSTRUCTION SET ARCHITECTURE

The Intel x86-64 CPU is a commercial CPU that implements an instruction set of many instructions. It has evolved from a 16-bit CPU to the 32- and 64-bit Intel CPU's found in many contemporary computers. In order to better understand how these instructions are executed, it is helpful to understand the basic organization of the external memory and internal memory of the CPU:

The x86 instruction set architecture is an example of a 2-Operand Machine Design. That is, the arithmetic/logic instructions that require two source operands require the programmer to provide these operands explicitly. The destination operand is implicit.

The instruction set of the x86-64 assembly language consists of 256 one-byte opcode instructions, 256 two-byte opcode instructions, and additional 3-byte opcodes. The instruction set for the x86-64 is very complex. This is a result of preserving backward compatibility with earlier x86 based processors which employed smaller registers. For this reason, instructions come in families and the version of instruction to use depends on the size of the destination. The x86-64 assembler is often able to infer the appropriate instruction in a given family from the destination operand. For this reason it is possible to use the name of the family rather than the instruction explicitly. However, for more advanced programming it is good practice to choose the instruction appropriate to the operands being employed.

The memory organization of the x86-64 hardware architecture is as follows:

1. The **external memory** of the CPU (Denoted in these notes by "`M`", potentially provides 2^{64} locations each identified by an address between 0000000000000000_{16} and $FFFFFFFFFFFFFFFF_{16}$. At each address it is possible to store a binary sequence of length 8. More concisely, the x86-64's external memory can be viewed as a linear array of a 2^{64} bytes.
   - **Byte Addressable Locations** : Every address identifies a storage location where an 8-bit binary sequence can be stored.
   - **Data Aligned According to Type**:
     - 64-bit data values are stored at addresses ending in 0x0 or ox8.
     - 32-bit data values are stored at addresses eding in 0x0, 0x4, 0x8, or 0xc.
     - 16-bit data values are stored at even addresses.
     - 8-bit values can be stored at any address
   - **Little Endian Byte Order**: Multibyte data is stored sequentially with the least significant byte "first"; that is, at the lowest address.
2. The **internal memory** of the CPU includes the following:
   (a) 16×64 **register file**:

   general purpose registers : rax, rbx, rcx, rdx

   argument registers : rsi, rdi

   stack registers : rbp, rsp

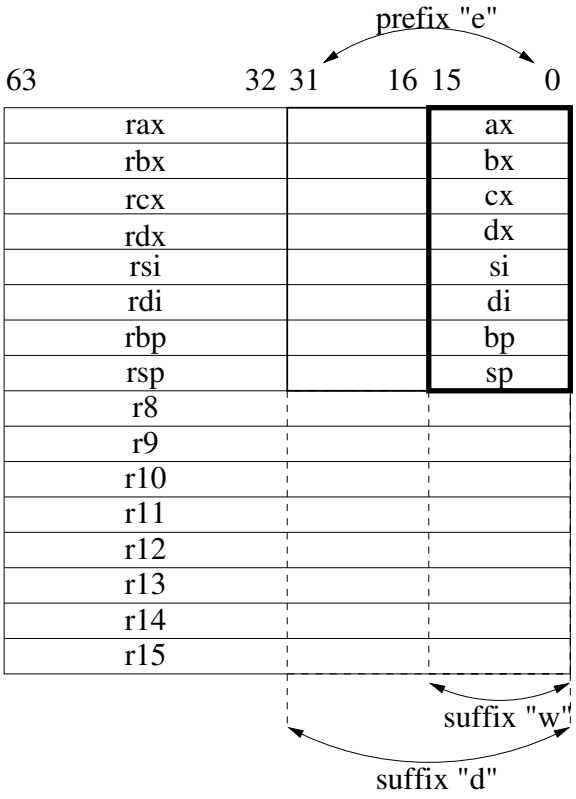
   supplementary registers : r8, r9, r10, r11, r12, r13, r14, r15

These 16 locations hold the operands for most arithmetic and logic instructions. Therefore it is the programmer's responsibility, using the instruction set, to transfer the values that are to serve as operands to these registers so that the operation defined by an arithmetic or logic instruction can be performed.

The locations of internal memory (i.e., the registers) are structured so that the architecture can model smaller hardware environments Each 64-bit register can also serve as a 32-bit, 16-bit or 8-bit register. The size of register used is determined by the name of the register. For example the first 64-bit work register in the register file is named "**rax**". However bits 31 down to 0 define the 32-bit register "**eax**". Similarly, bits 15 down to o define 16-bit register **ax**". Finally, bits 7 down to 0 define the 8-bit register named "**al**".

The following diagrams identifies the names of the registers of internal memory that can be referenced by the x86-64 instruction set.

The 16-bit, 32-bit, and 64-bit registers:



To summarize:

- The sixteen 64-bit registers each have a unique name, beginning with the letter "r'.
- The least significant 32 bits of each 64-bit register define a 32-bit register, each with a unique name beginning with "e".
- The least significant 16 bits of each 64-bit register define a 16-bit register, each with a unique name.

- The most significant and least significant 8 bits of each 16 bit register also define unique 8-bit registers:

	15 8	7 0
ax	ah	al
bx	bh	bl
cx	ch	cl
dx	dh	dl
si	sih	sil
di	dih	dil
sp	sph	spl
bp	bph	bpl

(b) **PC register**: This 64-bit register is the "Program Counter (PC)". Its purpose is to hold the address in memory of the next instruction to be executed. A "program" is a sequence of instructions stored in consecutive locations of memory. By incrementing this register each instruction can be retrieved in turn as the program is executed.

(c) **FLAGS Register**: The FLAGS register is used to store the result of comparing the values in two registers. Each bit has a unique name:

ZF : indicates whether the difference between the values being compared is zero; that is, the two values are equal.

SF : indicates whether the difference between the values is less than zero.

OF : indicates whether the difference results in an overflow.