Program Control: Recap

Condition Codes

In addition to integer registers, the CPU maintains a set of single-bit condition code registers. These registers can be tested to perform conditional branches. The most useful condition code registers are listed in the following table.

CC register	Name	Description
`CF`	Carry flag	The most recent operation generated a carry out of the most significant bit.
`ZF`	Zero flag	The most recent operation yielded 0.
`SF`	Sign flag	The most recent operation yielded a negative value.
`OF`	Overflow flag	The most recent operation caused a two's complement overflow (either negative or positive).

The *leaq* instruction does not alter any condition codes. Otherwise, all of the instructions listed in the ***Arithmetic and Logical Operations: Recap*** **(https://canvas.instructure.com/courses/1517115/pages/arithmetic-and-logical-operations-recap)** page cause the condition codes to be set. The logical operations clear *CF* and *OF* to 0, and set *ZF* and *SF* according to the result. The shift operations set *CF* to the last bit shifted out, set *ZF* and *SF* according to the result, and affect *OF* only for 1-bit shifts, otherwise it's undefined (I believe the book has got this part wrong). The *inc* and *dec* instructions set *SF*, *OF,* and *ZF* according to the result but leave *CF* unchanged.

In addition to the setting of condition codes by the instructions listed in the ***Arithmetic and Logical Operations: Recap*** **(https://canvas.instructure.com/courses/1517115/pages/arithmetic-and-logical-operations-recap)** , there are two instructions classes that set condition codes without altering any other registers.

Instruction	Based on	Description
CMP S_1, S_2	$S_2 - S_1$	Compare
`cmpb`		Compare byte
`cmpw`		Compare word

`cmpl`		Compare double word
`cmpq`		Compare quad word
TEST S_1, S_2	S_2 & S_1	Test
`testb`		Test byte
`testw`		Test word
`testl`		Test double word
`testq`		Test quad word

The *CMP* instructions set the condition codes according to the differences of their two operands. They set the condition codes in the same way as the corresponding *SUB* instructions, but *without* updating their destinations. With the assembler format we're using in this class, the operands are listed in reverse order, can make the code slightly awkward to read (i.e., the destination, which is also the first operand, appears on the right and the second operand appears on the left).

The *TEST* instructions set the condition codes in the same way as the corresponding *AND* instructions, but *without* updating their destinations. Typically, the same operand is repeated (e.g., *testq %rax, %rax* to see if *%rax* is negative, zero, or positive) or one operand is a mask indicating which bits should be tested.

Accessing the Condition Codes

Rather than reading the condition codes directly, there are three common ways of using the condition codes: (1) set a single byte to 0x00 or 0x01 depending on some combination of the condition codes, (2) jump conditionally to some other part of the program, or (3) transfer data conditionally.

Set a Single Byte

The *SET* instructions set a singe byte to 0x00 or 0x01 depending on some combination of the condition codes.

Instruction	**Synonym**	**Effect**	**Set condition**
`sete` *D*	`setz`	$D \leftarrow ZF$	Equal/zero
`setne` *D*	`setnz`	$D \leftarrow \sim ZF$	Not equal/not zero

`sets D`		$D \leftarrow SF$	Negative
`setns D`		$D \leftarrow \sim SF$	Nonnegative
`setg D`	`setnle`	$D \leftarrow \sim(SF \wedge OF)\ \&\ \sim ZF$	Greater (signed >)
`setge D`	`setnl`	$D \leftarrow \sim(SF \wedge OF)$	Greater or equal (signed ≥)
`setl D`	`setnge`	$D \leftarrow SF \wedge OF$	Less (signed <)
`setle D`	`setng`	$D \leftarrow (SF \wedge OF)\ \&\ \sim ZF$	Less or equal (signed ≤)
`seta D`	`setnbe`	$D \leftarrow \sim CF\ \&\ \sim ZF$	Above (unsigned >)
`setae D`	`setnb`	$D \leftarrow \sim CF$	Above or equal (unsigned ≥)
`setb D`	`setnae`	$D \leftarrow CF$	Below (unsigned <)
`setbe D`	`setna`	$D \leftarrow CF \mid ZF$	Below or equal (unsigned ≤)

Jump Instructions

Under normal execution, instructions follow each other in the order they are listed. A *jump* instruction can cause the execution to switch to a completely new position in the program.

Instruction	**Synonym**	**Jump condition**	**Description**
`jmp Label`		1 (always)	Direct jump
`jmp *Operand`		1 (always)	Indirection jump
`je Label`	`jz`	ZF	Equal/zero
`jne Label`	`jnz`	$\sim ZF$	Not equal/not zero
`js Label`		SF	Negative

`jns Label`		*~SF*	Nonnegative
`jg Label`	`jnle`	~(*SF* ^ *OF*) & ~*ZF*	Greater (signed >)
`jge Label`	`jnl`	~(*SF* ^ *OF*)	Greater or equal (signed ≥)
`jl Label`	`jnge`	*SF* ^ *OF*	Less (signed <)
`jle Label`	`jng`	(*SF* ^ *OF*) | *ZF*	Less or equal (signed ≤)
`ja Label`	`jnbe`	~*CF* & ~*ZF*	Above (unsigned >)
`jae Label`	`jnb`	~*CF*	Above or equal (unsigned ≥)
`jb Label`	`jnae`	*CF*	Below (unsigned <)
`jbe Label`	`jna`	*CF* | *ZF*	Below or equal (unsigned ≤)

The first two *jump* instruction are unconditional. The first one is a direct *jump* where the destination is encoded as part of the instruction. The second one is an indirect *jump* where the destination is read from a register (e.g., *jmp *%rax*) or a memory location (e.g., *jmp *(%rax)*). The remaining *jump* instructions are conditional; they either jump or continue executing at the next instruction depending on some combination of the condition codes.

There are two different encodings of for the jump targets: (1) PC relative and (2) absolute. In PC relative, the instruction encoding specifies the difference between the address of the target instruction and the address of the instruction immediately *following* the jump (owing to how early Intel processors were implemented). The difference is encoded in two's complement. In absolute, the instruction encoding specifies the actual address of the target instruction. The assembler and linker select the appropriate encodings of the jump destinations.

Conditional Move Instructions

Instruction	**Synonym**	**Move condition**	**Description**
`cmove S, R`	`cmovz`	*ZF*	Equal/zero
`cmovne S, R`	`cmovnz`	~*ZF*	Not equal/not zero

`cmovs S, R`	`cmovz`	*SF*	Negative
`cmovns S, R`	`cmovz`	~*SF*	Nonnegative
`cmovg S, R`	`cmovnle`	~(*SF* ^ *OF*) & ~*ZF*	Greater (signed >)
`cmovge S, R`	`cmovnl`	~(*SF* ^ *OF)*	Greater or equal (signed ≥)
`cmovl S, R`	`cmovnge`	*SF* ^ *OF*	Less (signed <)
`cmovle S, R`	`cmovng`	(*SF* ^ *OF*) | *ZF*	Less or equal (signed ≤)
`cmova S, R`	`cmovnbe`	~*CF* & ~*ZF*	Above (unsigned >)
`cmovae S, R`	`cmovnb`	~*CF*	Above or equal (unsigned ≥)
`cmovb S, R`	`cmovnae`	*CF*	Below (unsigned <)
`cmovbe S, R`	`cmovna`	*CF* | *ZF*	Below or equal (unsigned ≤)