THE UNIVERSITY OF ALABAMA AT BIRMINGHAM

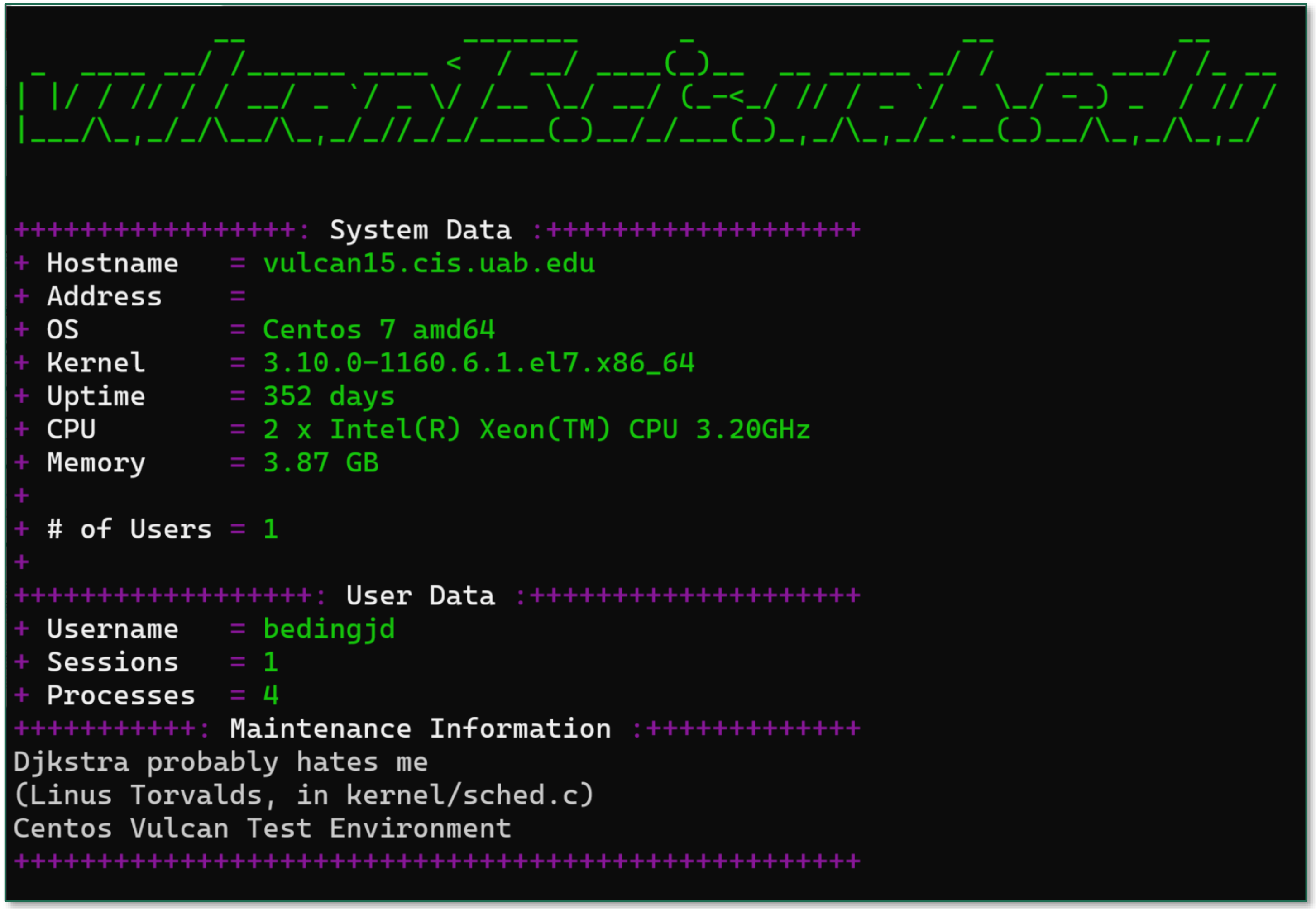
CS330

Program Control

Spring 2022

```
.section .data         # start of data section
# === global, static variables here ===

.section  .rodata      # start of read-only data section
# === constants here ===

.text                  # start of text / code
.global main           # tells computer we're starting at main

# === functions here ===

main:                  # start of main, required
# preamble
pushq %rbp
movq %rsp, %rbp

# code here


# return 0
movq $0, %rax          # move 0 into rax to return
leave                  # undo preamble
ret

```

Lab 11



Letters to the Editor

Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing
CR Categories: 4.22, 5.23, 5.24

EDITOR:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the "making" of the corresponding process is delegated to the machine.

My second remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.

Let us now consider how we can characterize the progress of a process. (You may think about this question in a very concrete manner: suppose that a process, considered as a time succession of actions, is stopped after an arbitrary action, what data do we have to fix in order that we can redo the process until the very same point?) If the program text is a pure concatenation of, say, assignment statements (for the purpose of this discussion regarded as the descriptions of single actions) it is sufficient to point in the program text to a point between two successive action descriptions. (In the absence of **go to** statements I can permit myself the syntactic ambiguity in the last three words of the previous sentence: if we parse them as "successive (action descriptions)" we mean successive in text space; if we parse as "(successive action) descriptions" we mean successive in time.) Let us call such a pointer to a suitable place in the text a "textual index."

When we include conditional clauses (**if** B **then** A), alternative clauses (**if** B **then** $A1$ **else** $A2$), choice clauses as introduced by C. A. R. Hoare (case[i] of $(A1, A2, \cdots, An)$), or conditional expressions as introduced by J. McCarthy ($B1 \rightarrow E1$, $B2 \rightarrow E2$, $\cdots$, $Bn \rightarrow En$), the fact remains that the progress of the process remains characterized by a single textual index.

As soon as we include in our language procedures we must admit that a single textual index is no longer sufficient. In the case that a textual index points to the interior of a procedure body the dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the dynamic depth of procedure calling.

Let us now consider repetition clauses (like, **while** B **repeat** A or **repeat** A **until** B). Logically speaking, such clauses are now superfluous, because we can express repetition with the aid of recursive procedures. For reasons of realism I don't wish to exclude them: on the one hand, repetition clauses can be implemented quite comfortably with present day finite equipment; on the other hand, the reasoning pattern known as "induction" makes us well equipped to retain our intellectual grasp on the processes generated by repetition clauses. With the inclusion of the repetition clauses textual indices are no longer sufficient to describe the dynamic progress of the process. With each entry into a repetition clause, however, we can associate a so-called "dynamic index," inexorably counting the ordinal number of the corresponding current repetition. As repetition clauses (just as procedure calls) may be applied nestedly, we find that now the progress of the process can always be uniquely characterized by a (mixed) sequence of textual and/or dynamic indices.

The main point is that the values of these indices are outside programmer's control; they are generated (either by the write-up of his program or by the dynamic evolution of the process) whether he wishes or not. They provide independent coordinates in which to describe the progress of the process.

Why do we need such independent coordinates? The reason is—and this seems to be inherent to sequential processes—that we can interpret the value of a variable only with respect to the progress of the process. If we wish to count the number, n say, of people in an initially empty room, we can achieve this by increasing n by one whenever we see someone entering the room. In the in-between moment that we have observed someone entering the room but have not yet performed the subsequent increase of n, its value equals the number of people in the room minus one!

The unbridled use of the **go to** statement has an immediate consequence that it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress. Usually, people take into account as well the values of some well chosen variables, but this is out of the question because it is relative to the progress that the meaning of these values is to be understood! With the **go to** statement one can, of course, still describe the progress uniquely by a counter counting the number of actions performed since program start (viz. a kind of normalized clock). The difficulty is that such a coordinate, although unique, is utterly unhelpful. In such a coordinate system it becomes an extremely complicated affair to define all those points of progress where, say, n equals the number of persons in the room minus one!

The **go to** statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program. One can regard and appreciate the clauses considered as bridling its use. I do not claim that the clauses mentioned are exhaustive in the sense that they will satisfy all needs, but whatever clauses are suggested (e.g. abortion clauses) they should satisfy the requirement that a programmer independent coordinate system can be maintained to describe the process in a helpful and manageable way.

It is hard to end this with a fair acknowledgment. Am I to

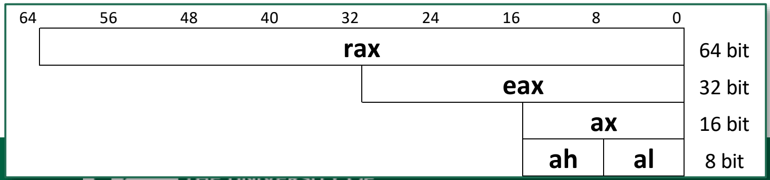
Djkstra, Go To Statement Considered Harmful

```c
static inline int goodness(struct task_struct * p, int this_cpu, struct mm_struct *this_mm)
{
        int weight;

        /*
         * select the current process after every other
         * runnable process, but before the idle thread.
         * Also, dont trigger a counter recalculation.
         */
        weight = -1;
        if (p->policy & SCHED_YIELD)
                goto out;

        /*
         * Non-RT process - normal case first.
         */
        if (p->policy == SCHED_OTHER) {
                /*
                 * Give the process a first-approximation goodness value
                 * according to the number of clock-ticks it has left.
                 *
                 * Don't do any other calculations if the time slice is
                 * over..
                 */
                weight = p->counter;
                if (!weight)
                        goto out;
```

How to read / interpret the syntax

- Typical AT&T mnemonics use three letter instructions with a one letter suffix to represent the size

Suffix		
b	byte	1 byte
w	word	2 bytes
l	doubleword	4 bytes
q	quadword	8 bytes



Instruction		Effect	Description	pg
Data Movement				
mov	S, D	$D \leftarrow S$	Move source to destination (movslq, sign extend l to q, pg 222)	183
push	S	$R[\%rsp] \leftarrow R[\%rsp] - 8$ $M[R[\%rsp]] \leftarrow S$	push source onto stack	189
pop	D	D $\leftarrow M[R[\%rsp]]$ $R[\%rsp] \leftarrow R[\%rsp] + 8$	pop top of stack into destination	189
Arithmetic				
lea	S, D	D $\leftarrow \&S$	load effective address	191
add	S, D	D $\leftarrow D + S$	add	192
sub	S, D	D $\leftarrow D - S$	subtract	192
mul	S, D	D $\leftarrow D * S$	multiply	192
imulq	S	R[%rdx]:R[%rax] $\leftarrow S * R[\%rax]$	multiply (2 64 bit numbers)	198
xor	S, D	D $\leftarrow D$^S	exclusive-or	192
cqto		R[%rdx]:R[%rax] $\leftarrow SignExtend(R[\%rax])$	Convert to oct word (sign extend)	198
idivq	S	$R[\%rdx] \leftarrow$ R[%rdx]:R[%rax] mod S $R[\%rax] \leftarrow$ R[%rdx]:R[%rax] / S	signed divide	198
Control				
cmp	S_1, S_2	$S_2 - S_1$	compare	202
jmp	label		direct jump	205
jmp	*Operand		indirect jump	205
je	label		jump if equal / zero (Zero Flag set)	205

S = Source, D = Destination

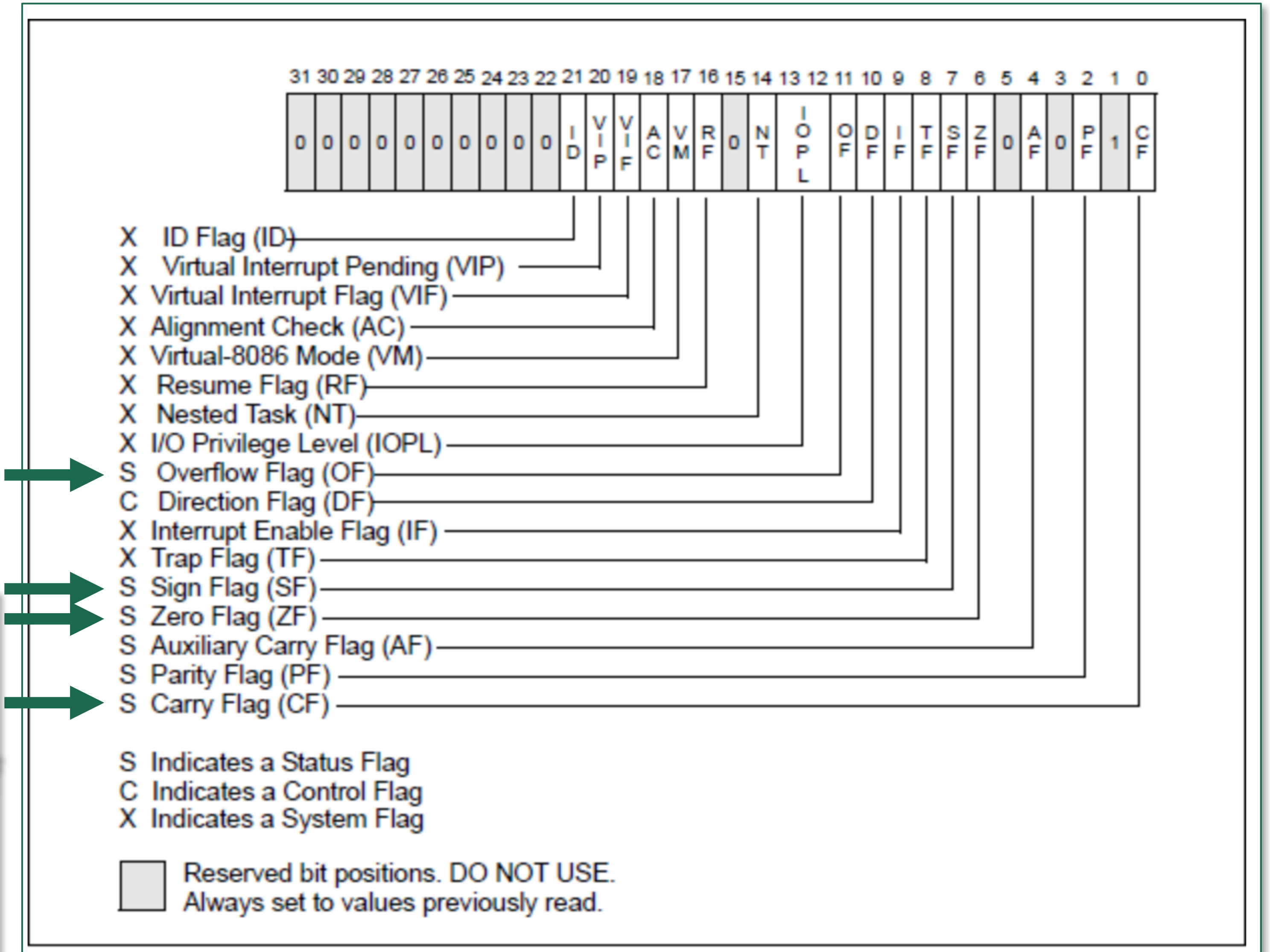
eflags
(and how to view registers)

- To view in GDB
  (i)nfo (r)egisters eflags

```
(gdb) info registers eflags
eflags         0x202      [ IF ]
```

  or
  tui reg general

- These are set:
  - **Implicitly** by arithmetic operations. Think of them as being a side effect of arithmetic operations
  - **Explicitly** by compare operations



EFLAGS Register

Condition Codes – Implicit Setting

- Example:

```
# add a+b = c
addq %rbx, %rax            # result, c in rax
```

- **CF Set** if carry occurs out from most significant bit
- **SF Set** if c < 0 (negative)
- **OF Set** if two's-complement signed arithmetic yields incorrect sign
- **ZF Set** if c == 0

NOT set by leaq instruction, even though leaq can be used in tricky ways to do math

Condition Code – Explicit Setting with Compare

- Example: $\mathrm{cmpq}\ \mathrm{src1},\ \mathrm{src2}$
- Same as computing src2 **–** src1 without setting a destination
  - Result is **not** stored, but flags are still set
- **CF Set ->** if carry occurs from most significant bit (leftmost)
- **ZF Set ->** if Src1 == Src2
- **OF Set ->** if overflow occurs
- **SF Set ->** if Src2 – Src1 < 0 (negative)

Condition Code – Explicit Setting with Test

- Example: testq src1, src2
- Same as computing src1 **&** src2 without setting a destination
  - Result is **not** stored, but flags are still set
  - Allows conditional statements on Boolean expressions
- **ZF Set ->** if Src1 & Src2 == 0
- **SF Set ->** if Src1 & Src2 < 0 (negative)

Jump Commands

Syntax:

- **Direct**:

  <jX> <label>

- **Indirect**:

  <jX> <*operand>

jX	Condition	Description
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	~ZF	Not Equal / Not Zero
js	SF	Negative
jns	~SF	Nonnegative
jg	~(SF^OF)&~ZF	Greater (Signed)
jge	~(SF^OF)	Greater or Equal (Signed)
jl	(SF^OF)	Less (Signed)
jle	(SF^OF)|ZF	Less or Equal (Signed)
ja	~CF&~ZF	Above (unsigned)
jb	CF	Below (unsigned)

An Example

- Let’s replicate the following

```c
int addif(int x, int y){
    int result = x + y;

    while(result <= 15){
        x++;
        y++;
        result = x + y;
    }
    return result;
}
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	Return value

Exercise to work – submit for attendance

- You can work in teams of 2
  - But everyone needs to submit to Canvas
- Write an assembly language program to print a star "*" pyramid:
  - Start with one star on a line
  - Print up to `n` stars

```
Please enter an int
5
*
**
***
****
*****
```

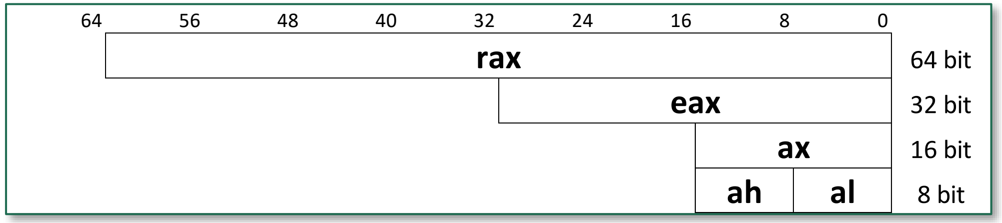
- Be sure to use at least one function
- You can take user input, or hardcode `n` in `main`
  - But need to pass it into the function(s)
- Print the pyramid

THE UNIVERSITY OF ALABAMA AT BIRMINGHAM

Reference

Registers

- 16 General Purpose Registers
- Register names per AT&T syntax
- Will not use floating, vector registers in this course
- Can also access subsets



Register	Usage	Old Names	Args	Saved by	Preserved Across Function Calls
%rax	temporary register; with variable arguments passes information about the number of vector registers used; **1st return register**	accumulator		**Caller**	No
%rbx	callee-saved register; optionally used as base pointer	base		**Callee**	Yes
%rcx	used to pass 4th integer argument to functions	counter, loop counter	4	**Caller**	No
%rdx	used to pass 3rd argument to functions; **2nd return register**	data	3	**Caller**	No
%rsp	stack pointer	stack pointer		**Callee**	Yes
%rbp	callee-aved register, optionally used as frame pointer	base pointer		**Callee**	Yes
%rsi	used to pass 2nd argument to functions	source index	**2**	**Caller**	No
%rdi	used to pass 1st argument to functions	destination index	**1**	**Caller**	No
%r8	used to pass 5th argument to functions		5	**Caller**	No
%r9	used to pass 6th argument to functions		6	**Caller**	No
%r10	temporary register, used for passing a function's static chain pointer			**Caller**	No
%r11	temporary register			**Caller**	No
%r12 - r15	callee-saved registers			**Callee**	Yes

GNU Assembler (AS) Manual: https://sourceware.org/binutils/docs/as/index.html#SEC_Contents

eflags
(and how to view registers)

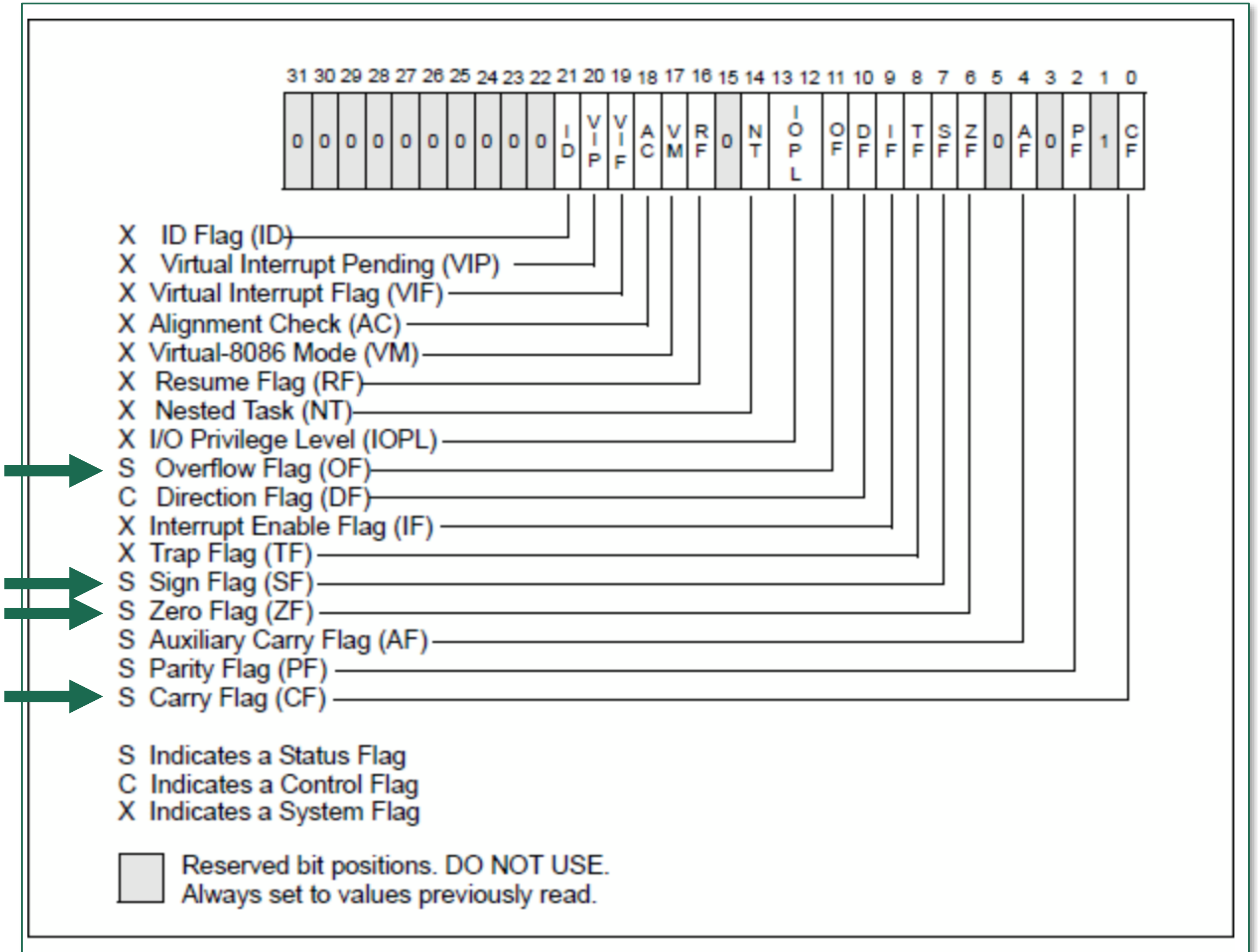
- In GDB
  (i)nfo (r)egisters eflags

```
(gdb) info registers eflags
eflags         0x202     [ IF ]
```

- To show all general purpose registers, including %rip (instruction pointer), eflags
  (i)nfo (r)egisters all

  or individually via
  (i)nfo (r)egisters $<name>
  e.g. (i)nfo (r)egisters $rax

  or
  tui reg general

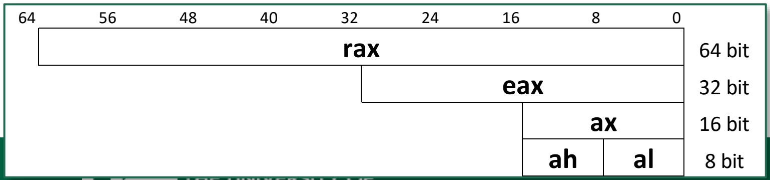


EFLAGS Register

How to read / interpret the syntax

- Typical AT&T mnemonics use three letter instructions with a one letter suffix to represent the size

Suffix		
b	byte	1 byte
w	word	2 bytes
l	doubleword	4 bytes
q	quadword	8 bytes



Instruction		Effect	Description	pg
Data Movement				
mov	S, D	$D \leftarrow S$	Move source to destination (movslq, sign extend l to q, pg 222)	183
push	S	$R[\%rsp] \leftarrow R[\%rsp] - 8$ $M[R[\%rsp]] \leftarrow S$	push source onto stack	189
pop	D	D $\leftarrow M[R[\%rsp]]$ $R[\%rsp] \leftarrow R[\%rsp] + 8$	pop top of stack into destination	189
Arithmetic				
lea	S, D	D $\leftarrow \&S$	load effective address	191
add	S, D	D $\leftarrow D + S$	add	192
sub	S, D	D $\leftarrow D - S$	subtract	192
mul	S, D	D $\leftarrow D * S$	multiply	192
imulq	S	R[%rdx]:R[%rax] $\leftarrow S * R[\%rax]$	multiply (2 64 bit numbers)	198
xor	S, D	D $\leftarrow D$^S	exclusive-or	192
cqto		R[%rdx]:R[%rax] ←SignExtend($R[\%rax]$)	Convert to oct word (sign extend)	198
idivq	S	$R[\%rdx] \leftarrow$ R[%rdx]:R[%rax] mod S $R[\%rax] \leftarrow$ R[%rdx]:R[%rax] / S	signed divide	198
Control				
cmp	S_1, S_2	S_2 - S_1	compare	202
jmp	label		direct jump	205
jmp	*Operand		indirect jump	205
je	label		jump if equal / zero (Zero Flag set)	205

S = Source, D = Destination

Operands take one of these three forms

1 Immediate / Literal: $4

2 Register: %rax

3 Memory

Type	From	Operand Value	Name
Immediate	$Imm	Imm	Immediate
Register	r_a	$R[r_a]$	Register
Memory	Imm	M[Imm]	Absolute
Memory	(r_a)	$M[R[r_a]]$	Indirect
Memory	$Imm(r_b)$	$M[Imm + R[r_b]]$	Base + displacement
Memory	$Imm(r_b, r_i, s)$	$M[Imm + R[r_b] + (R[r_i] * s)]$	Scaled Indexed

(see Book, pg 181 for more)

- Imm refers to a constant value, e.g. 0x8048d8e, 48
- r_a refers to a register
- $R[r_a]$ refers to the value stored in register r_a
- M[x] refers to the value stored at memory address x

Note: can’t move (mov) from Memory to Memory

Example assembly Instructions

Instruction		Effect	Description
`leaq`	S, D	$D \leftarrow \&S$	Load effective address
INC	D	$D \leftarrow D+1$	Increment
DEC	D	$D \leftarrow D-1$	Decrement
NEG	D	$D \leftarrow -D$	Negate
NOT	D	$D \leftarrow \sim D$	Complement
ADD	S, D	$D \leftarrow D + S$	Add
SUB	S, D	$D \leftarrow D - S$	Subtract
IMUL	S, D	$D \leftarrow D * S$	Multiply
XOR	S, D	$D \leftarrow D \hat{} S$	Exclusive-or
OR	S, D	$D \leftarrow D \mid S$	Or
AND	S, D	$D \leftarrow D \& S$	And
SAL	k, D	$D \leftarrow D << k$	Left shift
SHL	k, D	$D \leftarrow D << k$	Left shift (same as SAL)
SAR	k, D	$D \leftarrow D >>_A k$	Arithmetic right shift
SHR	k, D	$D \leftarrow D >>_L k$	Logical right shift

All of the instructions here are used for some kind of mathematical operation.
They show you the name of the instruction as it will be written in your code (but without the size- you may need to add the size suffix, such as q), and the order for the operands. S is Source, D is Destination.

Instruction		Effect	Description
`imulq`	S	R[%rdx]:R[%rax] $\leftarrow$ $S \times$ R[%rax]	Signed full multiply
`mulq`	S	R[%rdx]:R[%rax] $\leftarrow$ $S \times$ R[%rax]	Unsigned full multiply
`cqto`		R[%rdx]:R[%rax] $\leftarrow$ SignExtend(R[%rax])	Convert to oct word
`idivq`	S	R[%rdx] $\leftarrow$ R[%rdx]:R[%rax] mod S; R[%rax] $\leftarrow$ R[%rdx]:R[%rax] $\div$ S	Signed divide
`divq`	S	R[%rdx] $\leftarrow$ R[%rdx]:R[%rax] mod S; R[%rax] $\leftarrow$ R[%rdx]:R[%rax] $\div$ S	Unsigned divide

`**cqto**` has the purpose of sign-extending an integer in the %rax register to all of %rdx:%rax. For example, if %rax were –1 (11111111...), %rdx:%rax would be the same, but for all 128 bits.

The instructions here are different kinds of Multiplication and Division, except for cqto, which has a special purpose.
You will need to use Signed Multiplication and Division to see the correct results for all inputs on your homework, but positive inputs will behave the same way for both.

Carefully observe the "effect" column: Like Booth's algorithm, the multiplication result takes up twice as much space as the operands took up. Since we cannot know the operands are smaller than the size of the specified registers, the result is always stored across two whole registers.

`cltq` has the purpose of sign-extending an integer in the %eax register to all of %rax. For example, if %eax were –1 (11111111...), %rax would be the same, but for all 64 bits.

Functions

- Each function begins with a **label**:
  - A label is a name (capitalization matters) followed by a colon ":"
  - e.g. myFunction:
- Each function ends with a return **ret**
- Functions should be placed in the .text section, but above main:
- Don't forget our contract to pass arguments in the appropriate registers
  - Pass in arguments in the order: rdi, rsi, etc
  - Return values in rax
- Don't forget our contract to save appropriate registers
  - Ideally the caller-saved registers are the responsibility of the caller (e.g. main:), and the callee-saved registers are the responsibility of the callee (e.g. myFunction:)
  - Practically, since we're writing both the caller (main:) and the callee (myFunction:) functions, we can do whatever we want
    - And usually we don't waste resources saving registers unnecessarily, just the ones we need / use
- Functions can call other functions …
- **Be sure to document**, describe: what the function does, what it takes as arguments, what it returns