CMSC 313 COMPUTER ORGANIZATION & ASSEMBLY LANGUAGE PROGRAMMING

Lecture 6, Fall 2014

TOPICS TODAY

- Project Academic Integrity
- More on jump instructions
- Bit manipulation instructions
- More Arithmetic Instructions
 - NEG, MUL, IMUL, DIV
- Indexed Addressing Modes
- Some i386 String Instructions

SHORT VS NEAR JUMPS

SHORT JUMPS VS NEAR JUMPS

- Jumps use relative addressing
 - assembler computes an *offset* from address of current instruction.
 - produces *relocatable* code
- SHORT jumps use 8-bit offsets
 - target label within -128 bytes to +127 bytes
- NEAR jumps use 32-bit offsets
 - target label within -2³² bytes to +2³²-1 bytes

SHORT JUMPS VS NEAR JUMPS

- Some assemblers determine SHORT vs NEAR jumps automatically, but some do not.
- explicitly specify SHORT jumps
 jmp SHORT somewhere
- explicitly specify NEAR jumps

jge NEAR somewhere

```
; File: jmp.asm
; Demonstrating near and short jumps
;
       section .text
       global start
start: nop
       ; initialize
              eax, 17
                         ; eax := 17
start: mov
                          ; 17 - 42 is ...
              eax, 42
       cmp
                           ; exit if 17 >= 42
             exit
       jge
       jge short exit
       jge near exit
              exit
       jmp
       jmp
              short exit
              near exit
       jmp
              ebx, 0
                        ; exit code, 0=normal
exit:
      mov
              eax, 1
                          ; Exit.
       mov
              080н
                          ; Call kernel.
       int
```

1			; File:	jmp.asm						
2			;							
3			; Demon	strating	near	and short	jur	nps		
4			;							
5										
6				section	.text	t				
7				global _	_start	t				
8										
9	00000000	90	_start:	nop						
10					. .					
11				; initia	alize					
12										
13	00000001	B811000000	start:	mov	eax,	17	;	eax :	:= 17	
14	00000006	3D2A000000		cmp	eax,	42	;	17 -	42 1S	•••
15										
16	0000000B	7D14		jge	exit	_	;	exit	ıf 17	>= 42
17	000000D	7D12		jge	short	t exit				
18	000000F	0F8D0C000000		jge	near	exit				
19										
20	0000015	E907000000		jmp	exit					
21	000001A	EB05		jmp	short	t exit				
22	000001C	E900000000		jmp	near	exit				
23										
24	0000021	BB00000000	exit:	mov	ebx,	0	;	exit	code,	0=normal
25	0000026	B801000000		mov	eax,	1	;	Exit	•	
26	000002в	CD80		int	080н		;	Call	kernel	- •

BIT MANIPULATION

Logical (bit manipulation) Instructions

• AND: used to clear bits (store 0 in the bits):

 \diamond To clear the lower 4 bits of the AL register:

AND	AL, FOh	1101	0110
		<u>1111</u>	0000
		1101	0000

• OR: used to set bits (store 1 in the bits):

 \diamond To set the lower 4 bits of the AL register:

OR	AL, OFh	1101	0110
		0000	1111
		1101	1111

• NOT: flip all the bits

• Shift and Rotate instructions move bits around

INSTRUCTION SET REFERENCE

AND—Logical AND

Opcode	Instruction	Description
24 ib	AND AL, <i>imm</i> 8	AL AND imm8
25 iw	AND AX, <i>imm16</i>	AX AND imm16
25 id	AND EAX,imm32	EAX AND imm32
80 /4 <i>ib</i>	AND r/m8,imm8	r/m8 AND imm8
81 /4 <i>iw</i>	AND	r/m16 AND imm16
81 /4 <i>id</i>	AND <i>r/m32,imm32</i>	r/m32 AND imm32
83 /4 <i>ib</i>	AND r/m16,imm8	r/m16 AND imm8 (sign-extended)
83 /4 <i>ib</i>	AND r/m32,imm8	r/m32 AND imm8 (sign-extended)
20 /r	AND	<i>r/m</i> 8 AND <i>r</i> 8
21 /r	AND	<i>r/m16</i> AND <i>r16</i>
21 /r	AND <i>r/m32,r32</i>	r/m32 AND r32
22 /r	AND <i>r8,r/m8</i>	r8 AND r/m8
23 /r	AND <i>r16,r/m16</i>	r16 AND r/m16
23 /r	AND r32,r/m32	r32 AND r/m32

Description

Performs a bitwise AND operation on the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. (However, two memory operands cannot be used in one instruction.) Each bit of the result is set to 1 if both corresponding bits of the first and second operands are 1; otherwise, it is set to 0.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

Operation

DEST DEST AND SRC;

Flags Affected

The OF and CF flags are cleared; the SF, ZF, and PF flags are set according to the result. The state of the AF flag is undefined.

Protected Mode Exceptions

#GP(0)If the destination operand points to a nonwritable segment.If a memory operand effective address is outside the CS, DS, ES, FS, or
GS segment limit.If the DS, ES, FS, or GS register contains a null segment selector.

INSTRUCTION SET REFERENCE

Opcode	Instruction	Description
0C ib	OR AL, <i>imm8</i>	AL OR imm8
0D <i>iw</i>	OR AX,imm16	AX OR imm16
0D <i>id</i>	OR EAX,imm32	EAX OR imm32
80 /1 <i>ib</i>	OR	r/m8 OR imm8
81 /1 <i>iw</i>	OR r/m16,imm16	r/m16 OR imm16
81 /1 <i>id</i>	OR r/m32,imm32	r/m32 OR imm32
83 /1 <i>ib</i>	OR	r/m16 OR imm8 (sign-extended)
83 /1 <i>ib</i>	OR r/m32,imm8	r/m32 OR imm8 (sign-extended)
08 /r	OR <i>r/m8,r8</i>	r/m8 OR r8
09 / <i>r</i>	OR	r/m16 OR r16
09 /r	OR r/m32,r32	r/m32 OR r32
0A /r	OR r8,r/m8	r8 OR r/m8
0B /r	OR r16,r/m16	r16 OR r/m16
0B /r	OR r32,r/m32	r32 OR r/m32

OR—Logical Inclusive OR

Description

Performs a bitwise inclusive OR operation between the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. (However, two memory operands cannot be used in one instruction.) Each bit of the result of the OR instruction is set to 0 if both corresponding bits of the first and second operands are 0; otherwise, each bit is set to 1.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

Operation

DEST DEST OR SRC;

Flags Affected

The OF and CF flags are cleared; the SF, ZF, and PF flags are set according to the result. The state of the AF flag is undefined.

Protected Mode Exceptions

#GP(0)

If the destination operand points to a nonwritable segment.

If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register contains a null segment selector.

INSTRUCTION SET REFERENCE

NOT—One's Complement Negation

Opcode	Instruction	Description	
F6 /2	NOT <i>r/m8</i>	Reverse each bit of r/m8	
F7 /2	NOT <i>r/m16</i>	Reverse each bit of r/m16	
F7 /2	NOT <i>r/m32</i>	Reverse each bit of r/m32	

Description

Performs a bitwise NOT operation (each 1 is cleared to 0, and each 0 is set to 1) on the destination operand and stores the result in the destination operand location. The destination operand can be a register or a memory location.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

Operation

DEST NOT DEST;

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the destination operand points to a nonwritable segment.		
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.		
	If the DS, ES, FS, or GS register contains a null segment selector.		
#SS(0)	If a memory operand effective address is outside the SS segment limit.		
#PF(fault-code)	If a page fault occurs.		
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.		

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

INSTRUCTION SET REFERENCE

intel

SAL/SAR/SHL/SHR—Shift

Opcode	Instruction	Description
D0 /4	SAL <i>r/m</i> 8,1	Multiply r/m8 by 2, once
D2 /4	SAL r/m8,CL	Multiply r/m8 by 2, CL times
C0 /4 <i>ib</i>	SAL r/m8,imm8	Multiply r/m8 by 2, imm8 times
D1 /4	SAL <i>r/m16</i> ,1	Multiply r/m16 by 2, once
D3 /4	SAL	Multiply r/m16 by 2, CL times
C1 /4 <i>ib</i>	SAL r/m16,imm8	Multiply r/m16 by 2, imm8 times
D1 /4	SAL <i>r/m32</i> ,1	Multiply r/m32 by 2, once
D3 /4	SAL <i>r/m32</i> ,CL	Multiply r/m32 by 2, CL times
C1 /4 <i>ib</i>	SAL r/m32,imm8	Multiply r/m32 by 2, imm8 times
D0 /7	SAR <i>r/m8</i> ,1	Signed divide* r/m8 by 2, once
D2 /7	SAR r/m8,CL	Signed divide* r/m8 by 2, CL times
C0 /7 ib	SAR r/m8,imm8	Signed divide* r/m8 by 2, imm8 times
D1 /7	SAR <i>r/m16</i> ,1	Signed divide* r/m16 by 2, once
D3 /7	SAR r/m16,CL	Signed divide* r/m16 by 2, CL times
C1 /7 ib	SAR r/m16,imm8	Signed divide* r/m16 by 2, imm8 times
D1 /7	SAR <i>r/m32</i> ,1	Signed divide* r/m32 by 2, once
D3 /7	SAR <i>r/m32</i> ,CL	Signed divide* r/m32 by 2, CL times
C1 /7 ib	SAR r/m32,imm8	Signed divide* r/m32 by 2, imm8 times
D0 /4	SHL <i>r/m8</i> ,1	Multiply r/m8 by 2, once
D2 /4	SHL r/m8,CL	Multiply r/m8 by 2, CL times
C0 /4 <i>ib</i>	SHL r/m8,imm8	Multiply r/m8 by 2, imm8 times
D1 /4	SHL <i>r/m16</i> ,1	Multiply r/m16 by 2, once
D3 /4	SHL	Multiply r/m16 by 2, CL times
C1 /4 <i>ib</i>	SHL r/m16,imm8	Multiply r/m16 by 2, imm8 times
D1 /4	SHL <i>r/m32</i> ,1	Multiply r/m32 by 2, once
D3 /4	SHL r/m32,CL	Multiply r/m32 by 2, CL times
C1 /4 <i>ib</i>	SHL r/m32,imm8	Multiply r/m32 by 2, imm8 times
D0 /5	SHR <i>r/m8</i> ,1	Unsigned divide r/m8 by 2, once
D2 /5	SHR r/m8,CL	Unsigned divide r/m8 by 2, CL times
C0 /5 <i>ib</i>	SHR r/m8,imm8	Unsigned divide r/m8 by 2, imm8 times
D1 /5	SHR <i>r/m16</i> ,1	Unsigned divide r/m16 by 2, once
D3 /5	SHR r/m16,CL	Unsigned divide r/m16 by 2, CL times
C1 /5 <i>ib</i>	SHR r/m16,imm8	Unsigned divide <i>r/m16</i> by 2, <i>imm8</i> times
D1 /5	SHR <i>r/m</i> 32,1	Unsigned divide r/m32 by 2, once
D3 /5	SHR <i>r/m</i> 32,CL	Unsigned divide r/m32 by 2, CL times
C1 /5 <i>ib</i>	SHR r/m32,imm8	Unsigned divide <i>r/m32</i> by 2, <i>imm8</i> times

NOTE:

* Not the same form of division as IDIV; rounding is toward negative infinity.

SAL/SAR/SHL/SHR—Shift (Continued)

Description

Shifts the bits in the first operand (destination operand) to the left or right by the number of bits specified in the second operand (count operand). Bits shifted beyond the destination operand boundary are first shifted into the CF flag, then discarded. At the end of the shift operation, the CF flag contains the last bit shifted out of the destination operand.

The destination operand can be a register or a memory location. The count operand can be an immediate value or register CL. The count is masked to 5 bits, which limits the count range to 0 to 31. A special opcode encoding is provided for a count of 1.

The shift arithmetic left (SAL) and shift logical left (SHL) instructions perform the same operation; they shift the bits in the destination operand to the left (toward more significant bit locations). For each shift count, the most significant bit of the destination operand is shifted into the CF flag, and the least significant bit is cleared (see Figure 7-7 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*).

The shift arithmetic right (SAR) and shift logical right (SHR) instructions shift the bits of the destination operand to the right (toward less significant bit locations). For each shift count, the least significant bit of the destination operand is shifted into the CF flag, and the most significant bit is either set or cleared depending on the instruction type. The SHR instruction clears the most significant bit (see Figure 7-8 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*); the SAR instruction sets or clears the most significant bit to correspond to the sign (most significant bit) of the original value in the destination operand. In effect, the SAR instruction fills the empty bit position's shifted value with the sign of the unshifted value (see Figure 7-9 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*).

The SAR and SHR instructions can be used to perform signed or unsigned division, respectively, of the destination operand by powers of 2. For example, using the SAR instruction to shift a signed integer 1 bit to the right divides the value by 2.

Using the SAR instruction to perform a division operation does not produce the same result as the IDIV instruction. The quotient from the IDIV instruction is rounded toward zero, whereas the "quotient" of the SAR instruction is rounded toward negative infinity. This difference is apparent only for negative numbers. For example, when the IDIV instruction is used to divide -9 by 4, the result is -2 with a remainder of -1. If the SAR instruction is used to shift -9 right by two bits, the result is -3 and the "remainder" is +3; however, the SAR instruction stores only the most significant bit of the remainder (in the CF flag).

The OF flag is affected only on 1-bit shifts. For left shifts, the OF flag is cleared to 0 if the mostsignificant bit of the result is the same as the CF flag (that is, the top two bits of the original operand were the same); otherwise, it is set to 1. For the SAR instruction, the OF flag is cleared for all 1-bit shifts. For the SHR instruction, the OF flag is set to the most-significant bit of the original operand.

INSTRUCTION SET REFERENCE

SAL/SAR/SHL/SHR—Shift (Continued)

IA-32 Architecture Compatibility

The 8086 does not mask the shift count. However, all other IA-32 processors (starting with the Intel 286 processor) do mask the shift count to 5 bits, resulting in a maximum count of 31. This masking is done in all operating modes (including the virtual-8086 mode) to reduce the maximum execution time of the instructions.

Operation

```
tempCOUNT
               (COUNT AND 1FH);
tempDEST
            DEST;
WHILE (tempCOUNT
                     0)
DO
   IF instruction is SAL or SHL
       THEN
                 MSB(DEST);
           CF
       ELSE (* instruction is SAR or SHR *)
           CF
                 LSB(DEST);
   FI:
   IF instruction is SAL or SHL
       THEN
           DEST
                    DEST * 2;
       ELSE
           IF instruction is SAR
               THEN
                            DEST / 2 (*Signed divide, rounding toward negative infinity*);
                   DEST
               ELSE (* instruction is SHR *)
                            DEST / 2 ; (* Unsigned divide *);
                   DEST
           FI;
   FI;
   tempCOUNT
                  tempCOUNT - 1;
OD;
(* Determine overflow for the various instructions *)
IF COUNT
            1
   THEN
       IF instruction is SAL or SHL
           THEN
               OF
                    MSB(DEST) XOR CF;
           ELSE
               IF instruction is SAR
                   THEN
                       OF
                             0;
                   ELSE (* instruction is SHR *)
                       OF
                             MSB(tempDEST);
               FI;
       FI;
```

INSTRUCTION SET REFERENCE

SAL/SAR/SHL/SHR—Shift (Continued)

```
ELSE IF COUNT 0
THEN
All flags remain unchanged;
ELSE (* COUNT neither 1 or 0 *)
OF undefined;
FI;
```

Flags Affected

FI;

The CF flag contains the value of the last bit shifted out of the destination operand; it is undefined for SHL and SHR instructions where the count is greater than or equal to the size (in bits) of the destination operand. The OF flag is affected only for 1-bit shifts (see "Description" above); otherwise, it is undefined. The SF, ZF, and PF flags are set according to the result. If the count is 0, the flags are not affected. For a non-zero count, the AF flag is undefined.

Protected Mode Exceptions

#GP(0)	If the destination is located in a nonwritable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

Virtual-8086 Mode Exceptions			
#SS	If a memory operand effective address is outside the SS segment limit.		
#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.		

#GP(0)If a memory operand effective address is outside the CS, DS, ES, FS, or
GS segment limit.#SS(0)If a memory operand effective address is outside the SS segment limit.#PF(fault-code)If a page fault occurs.#AC(0)If alignment checking is enabled and an unaligned memory reference is
made.



After 1-bit SHL/SAL Instruction

After 10-bit SHL/SAL Instruction

0 🗲 0010001000100010001111000000000 🗲 0

Figure 7-7. SHL/SAL Instruction Operation



Figure 7-8. SHR Instruction Operation



Figure 7-9. SAR Instruction Operation

RCL/RCR/ROL/ROR--Rotate

Opcode	Instruction	Description
D0 /2	RCL <i>r/m8</i> , 1	Rotate 9 bits (CF, r/m8) left once
D2 /2	RCL <i>r/m8</i> , CL	Rotate 9 bits (CF, r/m8) left CL times
C0 /2 <i>ib</i>	RCL r/m8, imm8	Rotate 9 bits (CF, r/m8) left imm8 times
D1 /2	RCL <i>r/m16</i> , 1	Rotate 17 bits (CF, r/m16) left once
D3 /2	RCL <i>r/m16</i> , CL	Rotate 17 bits (CF, r/m16) left CL times
C1 /2 <i>ib</i>	RCL r/m16, imm8	Rotate 17 bits (CF, r/m16) left imm8 times
D1 /2	RCL <i>r/m32</i> , 1	Rotate 33 bits (CF, r/m32) left once
D3 /2	RCL <i>r/m</i> 32, CL	Rotate 33 bits (CF, r/m32) left CL times
C1 /2 <i>ib</i>	RCL r/m32,i mm8	Rotate 33 bits (CF, r/m32) left imm8 times
D0 /3	RCR <i>r/m8</i> , 1	Rotate 9 bits (CF, r/m8) right once
D2 /3	RCR <i>r/m8</i> , CL	Rotate 9 bits (CF, r/m8) right CL times
C0 /3 <i>ib</i>	RCR r/m8, imm8	Rotate 9 bits (CF, r/m8) right imm8 times
D1 /3	RCR <i>r/m16</i> , 1	Rotate 17 bits (CF, r/m16) right once
D3 /3	RCR <i>r/m16</i> , CL	Rotate 17 bits (CF, r/m16) right CL times
C1 /3 <i>ib</i>	RCR r/m16, imm8	Rotate 17 bits (CF, r/m16) right imm8 times
D1 /3	RCR <i>r/m32</i> , 1	Rotate 33 bits (CF, r/m32) right once
D3 /3	RCR <i>r/m32</i> , CL	Rotate 33 bits (CF, r/m32) right CL times
C1 /3 <i>ib</i>	RCR r/m32, imm8	Rotate 33 bits (CF, r/m32) right imm8 times
D0 /0	ROL <i>r/m8</i> , 1	Rotate 8 bits r/m8 left once
D2 /0	ROL <i>r/m8</i> , CL	Rotate 8 bits r/m8 left CL times
C0 /0 <i>ib</i>	ROL <i>r/m8, imm8</i>	Rotate 8 bits r/m8 left imm8 times
D1 /0	ROL <i>r/m16</i> , 1	Rotate 16 bits r/m16 left once
D3 /0	ROL <i>r/m16</i> , CL	Rotate 16 bits r/m16 left CL times
C1 /0 <i>ib</i>	ROL r/m16, imm8	Rotate 16 bits r/m16 left imm8 times
D1 /0	ROL <i>r/m32</i> , 1	Rotate 32 bits r/m32 left once
D3 /0	ROL <i>r/m32</i> , CL	Rotate 32 bits r/m32 left CL times
C1 /0 <i>ib</i>	ROL r/m32, imm8	Rotate 32 bits r/m32 left imm8 times
D0 /1	ROR <i>r/m8</i> , 1	Rotate 8 bits r/m8 right once
D2 /1	ROR <i>r/m8</i> , CL	Rotate 8 bits r/m8 right CL times
C0 /1 <i>ib</i>	ROR r/m8, imm8	Rotate 8 bits r/m16 right imm8 times
D1 /1	ROR <i>r/m16</i> , 1	Rotate 16 bits r/m16 right once
D3 /1	ROR <i>r/m16</i> , CL	Rotate 16 bits r/m16 right CL times
C1 /1 <i>ib</i>	ROR <i>r/m16, imm8</i>	Rotate 16 bits r/m16 right imm8 times
D1 /1	ROR <i>r/m</i> 32, 1	Rotate 32 bits r/m32 right once
D3 /1	ROR <i>r/m32</i> , CL	Rotate 32 bits r/m32 right CL times
C1 /1 <i>ib</i>	ROR <i>r/m32, imm8</i>	Rotate 32 bits r/m32 right imm8 times

RCL/RCR/ROL/ROR--Rotate (Continued)

Description

Shifts (rotates) the bits of the first operand (destination operand) the number of bit positions specified in the second operand (count operand) and stores the result in the destination operand. The destination operand can be a register or a memory location; the count operand is an unsigned integer that can be an immediate or a value in the CL register. The processor restricts the count to a number between 0 and 31 by masking all the bits in the count operand except the 5 least-significant bits.

The rotate left (ROL) and rotate through carry left (RCL) instructions shift all the bits toward more-significant bit positions, except for the most-significant bit, which is rotated to the least-significant bit location (see Figure 7-11 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*). The rotate right (ROR) and rotate through carry right (RCR) instructions shift all the bits toward less significant bit positions, except for the least-significant bit, which is rotated to the most-significant bit positions, except for the least-significant bit, which is rotated to the most-significant bit location (see Figure 7-11 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*).

The RCL and RCR instructions include the CF flag in the rotation. The RCL instruction shifts the CF flag into the least-significant bit and shifts the most-significant bit into the CF flag (see Figure 7-11 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*). The RCR instruction shifts the CF flag into the most-significant bit and shifts the least-significant bit into the CF flag (see Figure 7-11 in the *IA-32 Intel Architecture Software Developer's Manual, Volume 1*). For the ROL and ROR instructions, the original value of the CF flag is not a part of the result, but the CF flag receives a copy of the bit that was shifted from one end to the other.

The OF flag is defined only for the 1-bit rotates; it is undefined in all other cases (except that a zero-bit rotate does nothing, that is affects no flags). For left rotates, the OF flag is set to the exclusive OR of the CF bit (after the rotate) and the most-significant bit of the result. For right rotates, the OF flag is set to the exclusive OR of the two most-significant bits of the result.

IA-32 Architecture Compatibility

The 8086 does not mask the rotation count. However, all other IA-32 processors (starting with the Intel 286 processor) do mask the rotation count to 5 bits, resulting in a maximum count of 31. This masking is done in all operating modes (including the virtual-8086 mode) to reduce the maximum execution time of the instructions.

Operation

(* RCL and RCR instructions *) SIZE OperandSize CASE (determine count) OF SIZE 8: tempCOUNT (COUNT AND 1FH) MOD 9; SIZE 16: tempCOUNT (COUNT AND 1FH) MOD 17; SIZE 32: tempCOUNT COUNT AND 1FH; ESAC;

RCL/RCR/ROL/ROR--Rotate (Continued)

```
(* RCL instruction operation *)
WHILE (tempCOUNT 0)
  DO
      tempCF MSB(DEST);
      DEST (DEST * 2) + CF;
      CF tempCF;
      tempCOUNT tempCOUNT - 1;
  OD;
ELIHW;
IF COUNT 1
  THEN OF MSB(DEST) XOR CF;
  ELSE OF is undefined;
FI;
(* RCR instruction operation *)
IF COUNT 1
  THEN OF MSB(DEST) XOR CF;
  ELSE OF is undefined;
FI;
WHILE (tempCOUNT 0)
  DO
      tempCF LSB(SRC);
      DEST (DEST / 2) + (CF * 2^{SIZE});
      CF tempCF;
      tempCOUNT tempCOUNT - 1;
  OD:
(* ROL and ROR instructions *)
SIZE OperandSize
CASE (determine count) OF
  SIZE 8: tempCOUNT COUNT MOD 8;
  SIZE 16: tempCOUNT COUNT MOD 16;
  SIZE 32: tempCOUNT COUNT MOD 32;
ESAC;
(* ROL instruction operation *)
WHILE (tempCOUNT 0)
  DO
      tempCF MSB(DEST);
      DEST (DEST * 2) + tempCF;
      tempCOUNT tempCOUNT - 1;
  OD;
ELIHW;
CF LSB(DEST);
IF COUNT 1
  THEN OF MSB(DEST) XOR CF;
  ELSE OF is undefined;
FI;
```

3-662

INSTRUCTION SET REFERENCE

RCL/RCR/ROL/ROR--Rotate (Continued)

```
(* ROR instruction operation *)

WHILE (tempCOUNT 0)

DO

tempCF LSB(SRC);

DEST (DEST / 2) + (tempCF * 2<sup>SIZE</sup>);

tempCOUNT tempCOUNT - 1;

OD;

ELIHW;

CF MSB(DEST);

IF COUNT 1

THEN OF MSB(DEST) XOR MSB - 1(DEST);

ELSE OF is undefined;

FI;
```

Flags Affected

The CF flag contains the value of the bit shifted into it. The OF flag is affected only for singlebit rotates (see "Description" above); it is undefined for multi-bit rotates. The SF, ZF, AF, and PF flags are not affected.

Protected Mode Exceptions

#GP(0)	If the source operand is located in a nonwritable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.



Figure 7-11. ROL, ROR, RCL, and RCR Instruction Operations

Example using AND, OR & SHL

• Copy bits 4-7 of BX to bits 8-11 of AX

AX = 0110 1011 1001 0110 BX = 1101 0011 1100 0001

1. Clear bits 8-11 of AX & all but bits 4-7 of BX using AND instructions

AX	=	0110	0000	1001	0110	AND	AX,	FOFFh
BX	=	0000	0000	1100	0000	AND	BX,	00F0h

2. Shift bits 4-7 of BX to the desired position using a SHL instruction

AX	=	0110	0000	1001	0110			
BX	=	0000	1100	0000	0000	SHL	BX,	4

3. "Copy" bits of 4-7 of BX to AX using an OR instruction

AX = 0110 1100 1001 0110 OR AX, BX BX = 0000 1100 0000 0000

MORE ARITHMETIC INSTRUCTIONS

More Arithmetic Instructions

• NEG: two's complement negation of operand

• MUL: unsigned multiplication

- Multiply AL with r/m8 and store product in AX
- Multiply AX with r/m16 and store product in DX:AX
- O Multiply EAX with r/m32 and store product in EDX:EAX
- Immediate operands are not supported.
- \diamond CF and OF cleared if upper half of product is zero.

• IMUL: signed multiplication

- O Use with signed operands
- O More addressing modes supported

• DIV: unsigned division

INSTRUCTION SET REFERENCE

NEG—Two's Complement Negation

Opcode	Instruction	Description
F6 /3	NEG <i>r/m8</i>	Two's complement negate r/m8
F7 /3	NEG r/m16	Two's complement negate r/m16
F7 /3	NEG <i>r/m32</i>	Two's complement negate r/m32

Description

Replaces the value of operand (the destination operand) with its two's complement. (This operation is equivalent to subtracting the operand from 0.) The destination operand is located in a general-purpose register or a memory location.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

Operation

```
IF DEST 0
THEN CF 0
ELSE CF 1;
FI;
DEST – (DEST)
```

Flags Affected

The CF flag cleared to 0 if the source operand is 0; otherwise it is set to 1. The OF, SF, ZF, AF, and PF flags are set according to the result.

Protected Mode Exceptions

#GP(0)	If the destination is located in a nonwritable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Opcode	Instruction	Description
F6 /4	MUL r/m8	Unsigned multiply (AX AL * r/m8)
F7 /4	MUL r/m16	Unsigned multiply (DX:AX AX * r/m16)
F7 /4	MUL r/m32	Unsigned multiply (EDX:EAX EAX * r/m32)

MUL—Unsigned Multiply

Description

Performs an unsigned multiplication of the first operand (destination operand) and the second operand (source operand) and stores the result in the destination operand. The destination operand is an implied operand located in register AL, AX or EAX (depending on the size of the operand); the source operand is located in a general-purpose register or a memory location. The action of this instruction and the location of the result depends on the opcode and the operand size as shown in the following table.

Operand Size	Source 1	Source 2	Destination
Byte	AL	r/m8	AX
Word	AX	r/m16	DX:AX
Doubleword	EAX	r/m32	EDX:EAX

The result is stored in register AX, register pair DX:AX, or register pair EDX:EAX (depending on the operand size), with the high-order bits of the product contained in register AH, DX, or EDX, respectively. If the high-order bits of the product are 0, the CF and OF flags are cleared; otherwise, the flags are set.

Operation

```
IF byte operation

THEN

AX AL * SRC

ELSE (* word or doubleword operation *)

IF OperandSize 16

THEN

DX:AX AX * SRC

ELSE (* OperandSize 32 *)

EDX:EAX EAX * SRC

FI;

FI;
```

Flags Affected

The OF and CF flags are cleared to 0 if the upper half of the result is 0; otherwise, they are set to 1. The SF, ZF, AF, and PF flags are undefined.

INSTRUCTION SET REFERENCE

IMUL—Signed Multiply

Opcode	Instruction	Description
F6 /5	IMUL r/m8	AX AL * <i>r/m</i> byte
F7 /5	IMUL r/m16	DX:AX AX * r/m word
F7 /5	IMUL r/m32	EDX:EAX EAX * r/m doubleword
0F AF /r	IMUL r16,r/m16	word register word register * r/m word
0F AF /r	IMUL <i>r32,r/m32</i>	doubleword register doubleword register * r/m doubleword
6B /r ib	IMUL r16,r/m16,imm8	word register r/m16 * sign-extended immediate byte
6B /r ib	IMUL r32,r/m32,imm8	doubleword register r/m32 * sign-extended immediate byte
6B /r ib	IMUL r16,imm8	word register word register * sign-extended immediate byte
6B /r ib	IMUL r32,imm8	doubleword register doubleword register * sign- extended immediate byte
69 /r iw	IMUL	word register r/m16 * immediate word
69 /r id	IMUL r32,r/ m32,imm32	doubleword register r/m32 * immediate doubleword
69 /r iw	IMUL r16,imm16	word register r/m16 * immediate word
69 /r id	IMUL r32,imm32	doubleword register r/m32 * immediate doubleword

Description

Performs a signed multiplication of two operands. This instruction has three forms, depending on the number of operands.

- **One-operand form.** This form is identical to that used by the MUL instruction. Here, the source operand (in a general-purpose register or memory location) is multiplied by the value in the AL, AX, or EAX register (depending on the operand size) and the product is stored in the AX, DX:AX, or EDX:EAX registers, respectively.
- **Two-operand form.** With this form the destination operand (the first operand) is multiplied by the source operand (second operand). The destination operand is a general-purpose register and the source operand is an immediate value, a general-purpose register, or a memory location. The product is then stored in the destination operand location.
- **Three-operand form.** This form requires a destination operand (the first operand) and two source operands (the second and the third operands). Here, the first source operand (which can be a general-purpose register or a memory location) is multiplied by the second source operand (an immediate value). The product is then stored in the destination operand (a general-purpose register).

When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

INSTRUCTION SET REFERENCE

IMUL—Signed Multiply (Continued)

The CF and OF flags are set when significant bits are carried into the upper half of the result. The CF and OF flags are cleared when the result fits exactly in the lower half of the result.

The three forms of the IMUL instruction are similar in that the length of the product is calculated to twice the length of the operands. With the one-operand form, the product is stored exactly in the destination. With the two- and three- operand forms, however, result is truncated to the length of the destination before it is stored in the destination register. Because of this truncation, the CF or OF flag should be tested to ensure that no significant bits are lost.

The two- and three-operand forms may also be used with unsigned operands because the lower half of the product is the same regardless if the operands are signed or unsigned. The CF and OF flags, however, cannot be used to determine if the upper half of the result is non-zero.

Operation

```
IF (NumberOfOperands
                        1)
  THEN IF (OperandSize
                          8)
      THEN
          AX
               AL * SRC (* signed multiplication *)
          IF ((AH 00H) OR (AH
                                  FFH))
              THEN CF
                          0; OF
                                  0;
              ELSE CF
                          1; OF
                                  1;
          FI;
      ELSE IF OperandSize
                             16
          THEN
                       AX * SRC (* signed multiplication *)
              DX:AX
              IF ((DX
                       0000H) OR (DX FFFFH))
                  THEN CF
                              0; OF
                                     0;
                  ELSE CF
                              1; OF
                                      1;
              FI;
          ELSE (* OperandSize
                                 32 *)
                        EAX * SRC (* signed multiplication *)
              EDX:EAX
              IF ((EDX
                         00000000H) OR (EDX FFFFFFFH))
                  THEN CF
                              0; OF
                                      0;
                  ELSE CF
                              1; OF
                                      1;
              FI;
      FI;
  ELSE IF (NumberOfOperands
                                2)
      THEN
                  DEST * SRC (* signed multiplication; temp is double DEST size*)
          temp
          DEST
                   DEST * SRC (* signed multiplication *)
          IF temp DEST
              THEN CF
                          1; OF
                                  1;
              ELSE CF
                         0; OF
                                  0;
          FI;
      ELSE (* NumberOfOperands
                                   3*)
```

INSTRUCTION SET REFERENCE

IMUL—Signed Multiply (Continued)

```
DEST SRC1 * SRC2 (* signed multiplication *)

temp SRC1 * SRC2 (* signed multiplication; temp is double SRC1 size *)

IF temp DEST

THEN CF 1; OF 1;

ELSE CF 0; OF 0;

FI;

FI;

FI;
```

Flags Affected

For the one operand form of the instruction, the CF and OF flags are set when significant bits are carried into the upper half of the result and cleared when the result fits exactly in the lower half of the result. For the two- and three-operand forms of the instruction, the CF and OF flags are set when the result must be truncated to fit in the destination operand size and cleared when the result fits exactly in the destination operand size. The SF, ZF, AF, and PF flags are undefined.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
If a memory operand effective address is outside the SS segment limit.
If a page fault occurs.
If alignment checking is enabled and an unaligned memory reference is made.

DIV—Unsigned Divide

Opcode	Instruction	Description
F6 /6	DIV r/m8	Unsigned divide AX by $r/m8$, with result stored in AL \leftarrow Quotient, AH \leftarrow Remainder.
F7 /6	DIV <i>r/m16</i>	Unsigned divide DX:AX by $r/m16$, with result stored in AX \leftarrow Quotient, DX \leftarrow Remainder.
F7 /6	DIV <i>r/m</i> 32	Unsigned divide EDX:EAX by $r/m32$, with result stored in EAX \leftarrow Quotient, EDX \leftarrow Remainder.

Description

Divides (unsigned) the value in the AX, DX:AX, or EDX:EAX registers (dividend) by the source operand (divisor) and stores the result in the AX (AH:AL), DX:AX, or EDX:EAX registers. The source operand can be a general-purpose register or a memory location. The action of this instruction depends on the operand size (dividend/divisor). See Table 3-19.

Table 3-19. DIV Action

Operand Size	Dividend	Divisor	Quotient	Remainder	Maximum Quotient
Word/byte	AX	r/m8	AL	AH	255
Doubleword/word	DX:AX	r/m16	AX	DX	65,535
Quadword/doubleword	EDX:EAX	r/m32	EAX	EDX	$2^{32} - 1$

Non-integral results are truncated (chopped) towards 0. The remainder is always less than the divisor in magnitude. Overflow is indicated with the #DE (divide error) exception rather than with the CF flag.

Operation

```
 \begin{array}{l} \text{IF SRC} = 0 \\ \text{THEN \#DE; (* divide error *)} \\ \text{FI;} \\ \text{IF OperandSize} = 8 (* word/byte operation *) \\ \text{THEN} \\ temp \leftarrow AX / SRC; \\ \text{IF temp > FFH} \\ \text{THEN #DE; (* divide error *);} \\ \text{ELSE} \\ AL \leftarrow temp; \\ AH \leftarrow AX MOD SRC; \\ \text{FI;} \\ \text{ELSE} \\ \text{IF OperandSize} = 16 (* doubleword/word operation *) \\ \text{THEN} \end{array}
```

3-194 Vol. 2A

INSTRUCTION SET REFERENCE, A-M

```
temp \leftarrow DX:AX / SRC;
        IF temp > FFFFH
             THEN #DE; (* divide error *);
             ELSE
                  AX \leftarrow temp;
                  DX \leftarrow DX:AX MOD SRC;
        FI;
    ELSE (* quadword/doubleword operation *)
        temp \leftarrow EDX:EAX / SRC;
        IF temp > FFFFFFFH
             THEN #DE; (* divide error *);
             ELSE
                  EAX \leftarrow temp;
                  EDX ← EDX:EAX MOD SRC;
        FI;
FI;
```

Flags Affected

FI;

The CF, OF, SF, ZF, AF, and PF flags are undefined.

Protected Mode Exceptions

#DE	If the source operand (divisor) is 0				
	If the quotient is too large for the designated register.				
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.				
	If the DS, ES, FS, or GS register contains a null segment selector.				
#SS(0)	If a memory operand effective address is outside the SS segment limit.				
#PF(fault-code)	If a page fault occurs.				
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.				
Real-Address Mode Exceptions					
#DE	If the source operand (divisor) is 0.				
	If the quotient is too large for the designated register.				
#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.				
	If the DS, ES, FS, or GS register contains a null segment selector.				
#SS(0)	If a memory operand effective address is outside the SS segment limit.				

INSTRUCTION SET REFERENCE, A-M

Virtual-8086 Mode Exceptions

#DE	If the source operand (divisor) is 0.
	If the quotient is too large for the designated register.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

INDEXED ADDRESSING MODES

Indexed Addressing

- Operands of the form: [ESI + ECX*4 + DISP]
- ESI = Base Register
- ECX = Index Register
- 4 = Scale factor
- DISP = Displacement
- The operand is in memory
- The address of the memory location is ESI + ECX*4 + DISP



Figure 3-9. Offset (or Effective Address) Computation

The uses of general-purpose registers as base or index components are restricted in the following manner:

- The ESP register cannot be used as an index register.
- When the ESP or EBP register is used as the base, the SS segment is the default segment. In all other cases, the DS segment is the default segment.

The base, index, and displacement components can be used in any combination, and any of these components can be null. A scale factor may be used only when an index also is used. Each possible combination is useful for data structures commonly used by programmers in high-level languages and assembly language. The following addressing modes suggest uses for common combinations of address components.









Typical Uses for Indexed Addressing

• Base + Displacement

- \diamond access character in a string or field of a record
- o access a local variable in function call stack

• Index*Scale + Displacement

o access items in an array where size of item is 2, 4 or 8 bytes

• Base + Index + Displacement

access two dimensional array (displacement has address of array)

o access an array of records (displacement has offset of field in a record)

• Base + (Index*Scale) + Displacement

 \circ access two dimensional array where size of item is 2, 4 or 8 bytes

; File: index1.asm ; This program demonstrates the use of an indexed addressing mode ; to access array elements. ; ; This program has no I/O. Use the debugger to examine its effects. ; SECTION .data ; Data section dd 0, 1, 2, 3, 4, 5, 6, 7, 8, 9; ten 32-bit words arr: base: equ arr - 4 SECTION .text ; Code section. global start ; Entry point. start: nop ; Add 5 to each element of the array stored in arr. ; Simulate: for (i = 0; i < 10; i++) { ; arr[i] += 5 ; ; ; } init1: mov ecx, 0 ; ecx simulates i ecx, 10 ; i < 10 ? loop1: cmp jge done1 [ecx*4+arr], dword 5 ; arr[i] += 5 add inc ecx ; i++ jmp loop1 done1: ; more idiomatic for an assembly language program init2: mov ecx, 9 ; last array elt's index [ecx*4+arr], dword 5 loop2: add dec ecx jge loop2 ; again if ecx >= 0; another way ; base computed by ld edi, base init3: mov ; for(i=10 ; i>0 ; i--) ecx, 10mov [edi+ecx*4], dword 5 add loop3: loop loop3 ; loop = dec ecx, jne alldone: ebx, 0 ; exit code, 0=normal mov eax, 1 ; Exit. mov int 80H ; Call kernel.

Script started on Fri Sep 19 13:06:02 2003 linux3% nasm -f elf index1.asm linux3% ld index1.o linux3% gdb a.out GNU gdb Red Hat Linux (5.2-2) . . . (gdb) break *init1 Breakpoint 1 at 0x8048081 (gdb) break *init2 Breakpoint 2 at 0x8048099 (gdb) break *init3 Breakpoint 3 at 0x80480ac (gdb) break * alldone Breakpoint 4 at 0x80480bf (gdb) run Starting program: /afs/umbc.edu/users/c/h/chang/home/asm/a.out Breakpoint 1, 0x08048081 in init1 () (gdb) x/10wd &arr 1 0x80490cc <arr>: 0 2 3 7 0x80490dc <arr+16>: 4 5 6 0x80490ec <arr+32>: 8 9 (gdb) cont Continuing. Breakpoint 2, 0x08048099 in init2 () (qdb) x/10wd &arr 6 **0x80490cc** <arr>: 5 7 8 0x80490dc <arr+16>: 9 10 11 12 0x80490ec <arr+32>: 13 14 (gdb) cont Continuing. Breakpoint 3, 0x080480ac in init3 () (gdb) x/10wd &arr 0x80490cc <arr>: 10 11 12 13 0x80490dc <arr+16>: 14 15 16 17 0x80490ec <arr+32>: 18 19 (gdb) cont Continuing. Breakpoint 4, 0x080480bf in alldone () (gdb) x/10wd &arr 0x80490cc <arr>: 15 16 17 18 20 21 22 0x80490dc <arr+16>: 19 24 0x80490ec <arr+32>: 23 (gdb) cont Continuing. Program exited normally. (gdb) quit linux3% exit exit Script done on Fri Sep 19 13:07:41 2003

; File: index2.asm This program demonstrates the use of an indexed addressing mode ; ; to access 2 dimensional array elements. ; This program has no I/O. Use the debugger to examine its effects. ; SECTION .data ; Data section ; simulates a 2-dim array twodim: dd 00, 01, 02, 03, 04, 05, 06, 07, 08, 09 row1: dd 10, 11, 12, 13, 14, 15, 16, 17, 18, 19 row2: dd 20, 21, 22, 23, 24, 25, 26, 27, 28, 29 dd 30, 31, 32, 33, 34, 35, 36, 37, 38, 39 dd 40, 41, 42, 43, 44, 45, 46, 47, 48, 49 dd 50, 51, 52, 53, 54, 55, 56, 57, 58, 59 dd 60, 61, 62, 63, 64, 65, 66, 67, 68, 69 dd 70, 71, 72, 73, 74, 75, 76, 77, 78, 79 dd 80, 81, 82, 83, 84, 85, 86, 87, 88, 89 dd 90, 91, 92, 93, 94, 95, 96, 97, 98, 99 rowlen: equ row2 - row1 SECTION .text ; Code section. global _start start: nop ; Entry point. ; Add 5 to each element of row 7. Simulate: for (i = 0; i < 10; i++) { ; twodim[7][i] += 5 ; ; } ; init1: ecx, 0 ; ecx simulates i moveax, rowlen ; offset of twodim[7][0] mov edx, 7 mov mul edx ; eax := eax * edx jс alldone ; 64-bit product is bad ecx, 10 ; i < 10 ? loop1: cmp jge done1 add [eax+4*ecx+twodim], dword 5 ; i++ inc ecx jmp loop1 done1: alldone: mov ebx, 0 ; exit code, 0=normal mov eax, 1 ; Exit. 80H int ; Call kernel.

Script started on Fri Sep 19 13:19:22 2003 linux3% nasm -f elf index2.asm linux3% ld index2.0 linux3% linux3% gdb a.out GNU gdb Red Hat Linux (5.2-2) . . . (gdb) break *init1 Breakpoint 1 at 0x8048081 (gdb) break *alldone Breakpoint 2 at 0x80480a7 (qdb) run Starting program: /afs/umbc.edu/users/c/h/chang/home/asm/a.out Breakpoint 1, 0x08048081 in init1 () (gdb) x/10wd &twodim 0x80490b4 <twodim>: 0 1 2 3 0x80490c4 <twodim+16>: 4 5 7 6 0x80490d4 <twodim+32>: 9 8 (qdb) x/10wd &twodim+60 0x80491a4 <row2+200>: 60 61 62 63 0x80491b4 <row2+216>: 64 65 66 67 0x80491c4 <row2+232>: 68 69 (gdb) 70 0x80491cc <row2+240>: 71 72 73 75 0x80491dc <row2+256>: 74 76 77 0x80491ec <row2+272>: 78 79 (qdb) 0x80491f4 <row2+280>: 83 80 81 82 0x8049204 <row2+296>: 84 85 86 87 0x8049214 <row2+312>: 88 89 (gdb) cont Continuing. Breakpoint 2, 0x080480a7 in done1 () (gdb) x/10wd &twodim+60 0x80491a4 <row2+200>: 60 61 62 63 0x80491b4 <row2+216>: 64 65 66 67 0x80491c4 <row2+232>: 69 68 (gdb) 0x80491cc <row2+240>: 75 76 77 78 79 0x80491dc <row2+256>: 80 81 82 0x80491ec <row2+272>: 83 84 (gdb) 0x80491f4 <row2+280>: 80 81 82 83 0x8049204 <row2+296>: 85 84 86 87 0x8049214 <row2+312>: 89 88 (gdb) cont Continuing. Program exited normally. (gdb) quit linux3% exit exit Script done on Fri Sep 19 13:20:35 2003

NEXT TIME

Subroutines

References

• Some figures and diagrams from IA-32 Intel Architecture Software Developer's Manual, Vols 1-3

<http://developer.intel.com/design/Pentium4/manuals/>