CSSE132
Introduction to Computer Systems

14 : Arithmetic & Logic Operations

March 26, 2013

Today: Arithmetic & Logic Operations

- **Review addressing modes**
- **Movement instructions**
- **Arithmetic & logic operations**
- **Condition codes**
  - Compare and Test

Addressing modes

- **Move data from**
  - Immediate
  - Register
  - Memory
- **Move data to**
  - Register
  - Memory

- **Only 1 memory operation per instruction**

Addressing modes

Type	Form	Operand value	Name
Immediate	$\$Imm$	Imm	Immediate
Register	E_a	$\mathbf{R}[\mathrm{E}_a]$	Register
Memory	`Imm`	$\mathbf{M}[Imm]$	Absolute
Memory	(E_a)	$\mathbf{M}[\mathbf{R}[\mathrm{E}_a]]$	Indirect
Memory	$Imm(\mathrm{E}_b)$	$\mathbf{M}[Imm + \mathbf{R}[\mathrm{E}_b]]$	Base + displacement
Memory	$(\mathrm{E}_b, \mathrm{E}_i)$	$\mathbf{M}[\mathbf{R}[\mathrm{E}_b] + \mathbf{R}[\mathrm{E}_i]]$	Indexed
Memory	$Imm(\mathrm{E}_b, \mathrm{E}_i)$	$\mathbf{M}[Imm + \mathbf{R}[\mathrm{E}_b] + \mathbf{R}[\mathrm{E}_i]]$	Indexed
Memory	$(, \mathrm{E}_i, s)$	$\mathbf{M}[\mathbf{R}[\mathrm{E}_i] \cdot s]$	Scaled indexed
Memory	$Imm(, \mathrm{E}_i, s)$	$\mathbf{M}[Imm + +\mathbf{R}[\mathrm{E}_i] \cdot s]$	Scaled indexed
Memory	$(\mathrm{E}_b, \mathrm{E}_i, s)$	$\mathbf{M}[\mathbf{R}[\mathrm{E}_b] + \mathbf{R}[\mathrm{E}_i] \cdot s]$	Scaled indexed
Memory	$Imm(\mathrm{E}_b, \mathrm{E}_i, s)$	$\mathbf{M}[Imm + \mathbf{R}[\mathrm{E}_b] + \mathbf{R}[\mathrm{E}_i] \cdot s]$	Scaled indexed

Complete Memory Addressing Modes

- **Most General Form**
- **D(Rb,Ri,S)   Mem[Reg[Rb]+S*Reg[Ri]+ D]**
  - D: Constant "displacement" 1, 2, or 4 bytes
  - Rb: Base register: Any of 8 integer registers
  - Ri: Index register: Any, except for `%esp`
    - Unlikely you'd use `%ebp`, either
  - S: Scale: 1, 2, 4, or 8 (why these numbers?)

Address Computation Examples

%edx	0xf000
%ecx	0x0100

Expression	Address Computation	Address
0x8(%edx)		
(%edx,%ecx)		
(%edx,%ecx,4)		
0x80(,%edx,2)		

Address Computation Instruction

- **`leal` Src,Dest**
  - Src is address mode expression
  - Set Dest to address denoted by expression

- **Uses**
  - Computing addresses without a memory reference
    - E.g., translation of `p = &x[i];`
  - Computing arithmetic expressions of the form x + k*y
    - k = 1, 2, 4, or 8

- **Example**

```
int mul12(int x)
{
  return x*12;
}
```

Converted to ASM by compiler:

```
leal (%eax,%eax,2), %eax   ;t <- x+x*2
sall $2, %eax              ;return t<<2
```

Today: Arithmetic & Logic Operations

- **Review addressing modes**
- **Movement instructions**
- **Arithmetic & logic operations**
- **Condition codes**
  - Compare and Test

Movement instructions

- **mov instruction**
- **Move data between operand locations**
  - Supports byte, word, and double-word sizes
  - Can move from small size to larger size
    - Must sign-extend or zero-extend

- **pushl/popl**
- **Push & pop to stack**
  - Increase/decrease stack
  - Store/retrieve value on stack

Data movement

Instruction		Effect	Description
`mov`	S, D	$D \leftarrow S$	Move
`movb`		Move byte	
`movw`		Move word	
`movl`		Move double word	
`movs`	S, D	$D \leftarrow \texttt{SignExtend}(S)$	Move with sign extension
`movsbw`		Move sign-extended byte to word	
`movsbl`		Move sign-extended byte to double word	
`movswl`		Move sign-extended word to double word	
`movz`	S, D	$D \leftarrow \texttt{ZeroExtend}(S)$	Move with zero extension
`movzbw`		Move zero-extended byte to word	
`movzbl`		Move zero-extended byte to double word	
`movzwl`		Move zero-extended word to double word	
`pushl`	S	$\mathbf{R}[\%\texttt{esp}] \leftarrow \mathbf{R}[\%\texttt{esp}] - 4;$ $\mathbf{M}[\mathbf{R}[\%\texttt{esp}]] \leftarrow S$	Push double word
`popl`	D	$D \leftarrow \mathbf{M}[\mathbf{R}[\%\texttt{esp}]];$ $\mathbf{R}[\%\texttt{esp}] \leftarrow \mathbf{R}[\%\texttt{esp}] + 4$	Pop double word

Today: Arithmetic & Logic Operations

- **Review addressing modes**
- **Movement instructions**
- **Arithmetic & logic operations**
- **Condition codes**
  - Compare and Test

Some Arithmetic Operations

- **Two Operand Instructions:**

Format		Computation	
`addl`	Src,Dest	Dest = Dest + Src	
`subl`	Src,Dest	Dest = Dest - Src	
`imull`	Src,Dest	Dest = Dest * Src	
`sall`	Src,Dest	Dest = Dest << Src	Also called shll
`sarl`	Src,Dest	Dest = Dest >> Src	Arithmetic
`shrl`	Src,Dest	Dest = Dest >> Src	Logical
`xorl`	Src,Dest	Dest = Dest ^ Src	
`andl`	Src,Dest	Dest = Dest & Src	
`orl`	Src,Dest	Dest = Dest | Src	

- **Watch out for argument order!**
- **No distinction between signed and unsigned int (why?)**

Some Arithmetic Operations

- **One Operand Instructions**

`incl`	Dest	Dest = Dest + 1
`decl`	Dest	Dest = Dest - 1
`negl`	Dest	Dest = - Dest
`notl`	Dest	Dest = ~Dest

Arithmetic & Logic Operations

Instruction		Effect	Description
`leal`	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	$D \leftarrow D$ ^ S	Exclusive-ir
`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
`sar`	k, D	$D \leftarrow D >>_A k$	Arithmetic right shift
`shr`	k, D	$D \leftarrow D >>_L k$	Logical right shift

Arithmetic Expression Example

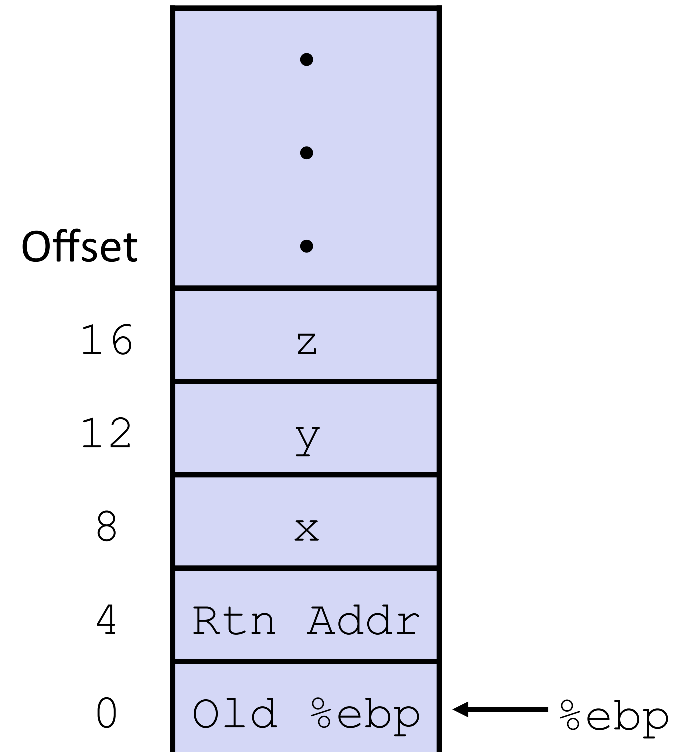
```c
int arith(int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```

```
arith:
  pushl  %ebp                   } Set Up
  movl   %esp, %ebp

  movl   8(%ebp), %ecx          } Body
  movl   12(%ebp), %edx
  leal   (%edx,%edx,2), %eax
  sall   $4, %eax
  leal   4(%ecx,%eax), %eax
  addl   %ecx, %edx
  addl   16(%ebp), %edx
  imull  %edx, %eax

  popl   %ebp                   } Finish
  ret
```

Understanding `arith`

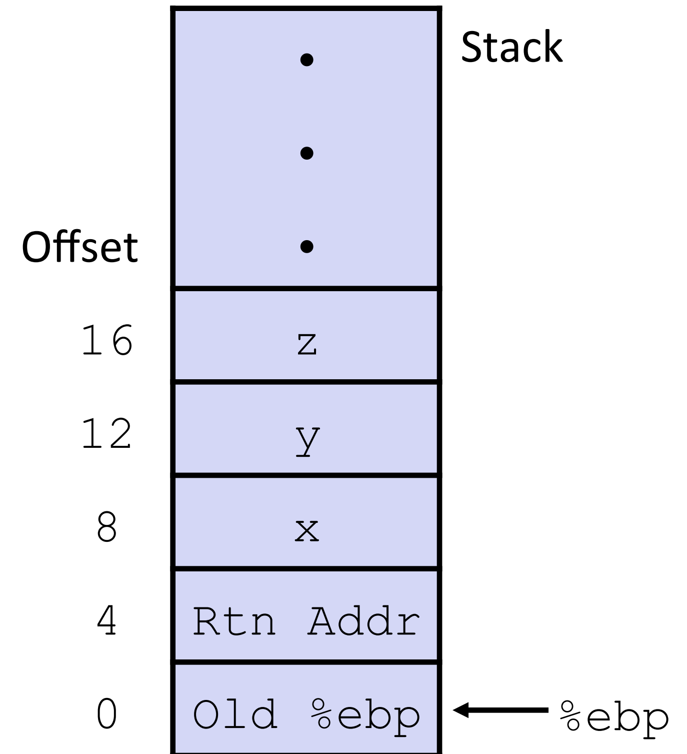
```
int arith(int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```

```
movl   8(%ebp), %ecx
movl   12(%ebp), %edx
leal   (%edx,%edx,2), %eax
sall   $4, %eax
leal   4(%ecx,%eax), %eax
addl   %ecx, %edx
addl   16(%ebp), %edx
imull  %edx, %eax
```



Understanding `arith`

```
int arith(int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```




```
movl   8(%ebp), %ecx          # ecx = x
movl   12(%ebp), %edx         # edx = y
leal   (%edx,%edx,2), %eax    # eax = y*3
sall   $4, %eax               # eax *= 16 (t4)
leal   4(%ecx,%eax), %eax     # eax = t4 +x+4 (t5)
addl   %ecx, %edx             # edx = x+y (t1)
addl   16(%ebp), %edx         # edx += z (t2)
imull  %edx, %eax             # eax = t2 * t5 (rval)
```

Observations about `arith`

```
int arith(int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```

- Instructions in different order from C code
- Some expressions require multiple instructions
- Some instructions cover multiple expressions
- Get exact same code when compile:
- `(x+y+z)*(x+4+48*y)`

```
movl   8(%ebp), %ecx           # ecx = x
movl   12(%ebp), %edx          # edx = y
leal   (%edx,%edx,2), %eax     # eax = y*3
sall   $4, %eax                # eax *= 16 (t4)
leal   4(%ecx,%eax), %eax      # eax = t4 +x+4 (t5)
addl   %ecx, %edx              # edx = x+y (t1)
addl   16(%ebp), %edx          # edx += z (t2)
imull  %edx, %eax              # eax = t2 * t5 (rval)
```

Another Example

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

```
logical:
    pushl %ebp
    movl %esp,%ebp

    movl 12(%ebp),%eax
    xorl 8(%ebp),%eax
    sarl $17,%eax
    andl $8185,%eax

    popl %ebp
    ret
```

Set Up (pushl, movl %esp,%ebp); Body (movl through andl); Finish (popl, ret)

```
movl 12(%ebp),%eax     # eax = y
xorl 8(%ebp),%eax      # eax = x^y         (t1)
sarl $17,%eax          # eax = t1>>17      (t2)
andl $8185,%eax        # eax = t2 & mask   (rval)
```

Another Example

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

```
logical:
    pushl %ebp            } Set
    movl %esp,%ebp        } Up

    movl 12(%ebp),%eax    }
    xorl 8(%ebp),%eax     } Body
    sarl $17,%eax         }
    andl $8185,%eax       }

    popl %ebp             } Finish
    ret                   }
```

```
movl 12(%ebp),%eax      # eax = y
xorl 8(%ebp),%eax       # eax = x^y        (t1)
sarl $17,%eax           # eax = t1>>17     (t2)
andl $8185,%eax         # eax = t2 & mask (rval)
```

Another Example

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

```
logical:
    pushl %ebp
    movl %esp,%ebp

    movl 12(%ebp),%eax
    xorl 8(%ebp),%eax
    sarl $17,%eax
    andl $8185,%eax

    popl %ebp
    ret
```

Set Up: pushl %ebp / movl %esp,%ebp

Body: movl 12(%ebp),%eax … andl $8185,%eax

Finish: popl %ebp / ret

```
movl 12(%ebp),%eax     # eax = y
xorl 8(%ebp),%eax      # eax = x^y          (t1)
sarl $17,%eax          # eax = t1>>17       (t2)
andl $8185,%eax        # eax = t2 & mask (rval)
```

Another Example

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

```
logical:
    pushl %ebp
    movl %esp,%ebp

    movl 12(%ebp),%eax
    xorl 8(%ebp),%eax
    sarl $17,%eax
    andl $8185,%eax

    popl %ebp
    ret
```

Set Up

Body

Finish

$2^{13} = 8192$, $2^{13} - 7 = 8185$

```
movl 12(%ebp),%eax     # eax = y
xorl 8(%ebp),%eax      # eax = x^y        (t1)
sarl $17,%eax          # eax = t1>>17     (t2)
andl $8185,%eax        # eax = t2 & mask  (rval)
```

Special Arithmetic Operations

- Multiply and divide can operate in more than 32 bits

- imull : full signed multiply
- mull : full unsigned multiply
  - Result is quadword product in %edx:%eax
- cltd : sign extend %eax into %edx (for full divide)
- idivl : full signed divide
- divl : full unsigned divide
  - Operates on quadword dividend in %edx:%eax
  - Result is quotient in %eax, remainder in %edx

Today: Arithmetic & Logic Operations

- Review addressing modes
- Movement instructions
- Arithmetic & logic operations
- Condition codes
  - Compare and Test

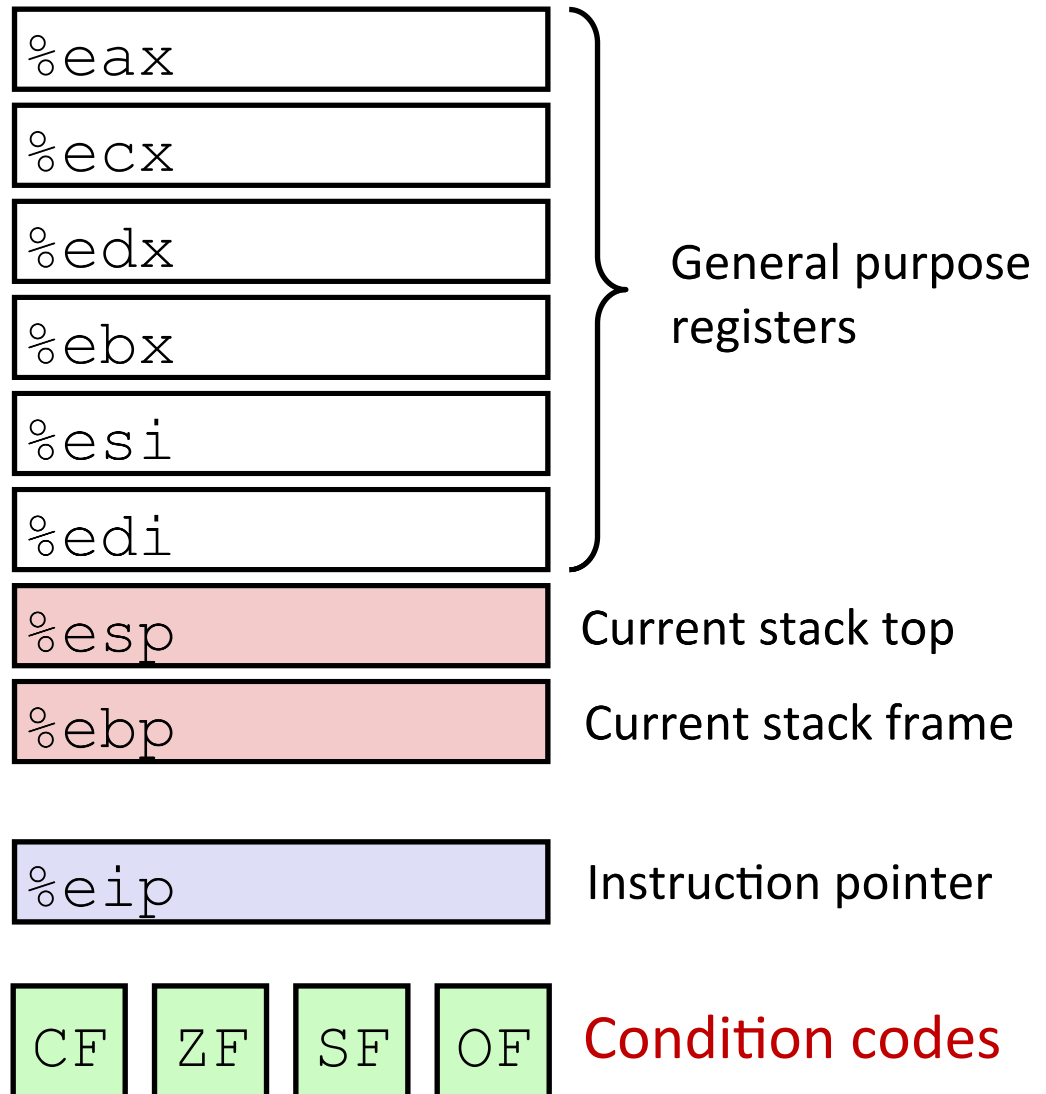
Condition codes

- Stored inside CPU
  - 8 Registers (32 bit)
  - PC register (32 bit)
  - 4 Condition codes (1 bit)

- Condition codes set on arithmetic operations
  - Automatically set
  - Or generated compare/test instructions
- Can be read indirectly
- Can be used to set up branching code
  - if, else, loops
  - Conditional data transfer

Processor State (IA32, Partial)

- **Information about currently executing program**
  - Temporary data
    (%eax, …)
  - Location of runtime stack
    (%ebp,%esp)
  - Location of current code control point
    (%eip, …)
  - Status of recent tests
    (CF, ZF, SF, OF)



Condition Codes (Implicit Setting)

- **Single bit registers**
  - CF Carry Flag (for unsigned) SF Sign Flag (for signed)
  - ZF Zero Flag OF Overflow Flag (for signed)
- **Implicitly set (think of it as side effect) by arithmetic operations**

  Example: `addl/addq` Src,Dest ↔ `t = a+b`

  CF set if carry out from most significant bit (unsigned overflow)

  ZF set if `t == 0`

  SF set if `t < 0` (as signed)

  OF set if two's-complement (signed) overflow
  `(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)`
- **Not set by `lea` instruction**
- **Full documentation (IA32), link on course website**

Condition Codes (Explicit Setting: Compare)

- **Explicit Setting by Compare Instruction**
  - `cmpl/cmpq` Src2, Src1
  - `cmpl b,a` like computing `a-b` without setting destination

  - CF set if carry out from most significant bit (used for unsigned comparisons)
  - ZF set if `a == b`
  - SF set if `(a-b) < 0` (as signed)
  - OF set if two's-complement (signed) overflow

```
(a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)
```

Condition Codes (Explicit Setting: Test)

- **Explicit Setting by Test instruction**
  - `testl`/`testq` Src2, Src1

  `testl b,a` like computing `a&b` without setting destination

  - Sets condition codes based on value of Src1 & Src2
  - Useful to have one of the operands be a mask

  - ZF set when `a&b == 0`
  - SF set when `a&b < 0`

Setting the Condition Codes

Instruction		Based on	Description
`cmp`	S_2, S_1	$S_1 - S_2$	Compare
`cmpb`		Compare byte	
`cmpw`		Compare word	
`cmpl`		Compare double word	
`test`	S_2, S_1	S_1 & S_2	Test
`testb`		Test byte	
`testw`		Test word	
`testl`		Test double word	

Reading Condition Codes

- **SetX Instructions**
  - Set single byte based on combinations of condition codes

SetX	Condition	Description
`sete`	`ZF`	Equal / Zero
`setne`	`~ZF`	Not Equal / Not Zero
`sets`	`SF`	Negative
`setns`	`~SF`	Nonnegative
`setg`	`~(SF^OF)&~ZF`	Greater (Signed)
`setge`	`~(SF^OF)`	Greater or Equal (Signed)
`setl`	`(SF^OF)`	Less (Signed)
`setle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`seta`	`~CF&~ZF`	Above (unsigned)
`setb`	`CF`	Below (unsigned)

Reading Codes (details)

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^OF)& $\sim$ ZF	Greater (signed >)
setge	D	setnl	$D \leftarrow \sim$ (SF^OF)	Greater or equal (signed >=)
setl	D	setnge	$D \leftarrow$ SF^OF	Less (signed <)
setle	D	setng	$D \leftarrow$ (SF^OF)$\mid$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 <=)

Reading Condition Codes (Cont.)

- **SetX Instructions:**
  - Set single byte based on combination of condition codes
- **One of 8 addressable byte registers**
  - Does not alter remaining 3 bytes
  - Typically use `movzbl` to finish job

```
int gt (int x, int y)
{
  return x > y;
}
```

Body

```
movl 12(%ebp),%eax   # eax = y
cmpl %eax,8(%ebp)    # Compare x : y
setg %al             # al = x > y
movzbl %al,%eax      # Zero rest of %eax
```

%eax	%ah	%al
%ecx	%ch	%cl
%edx	%dh	%dl
%ebx	%bh	%bl
%esi		
%edi		
%esp		
%ebp		