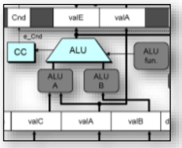
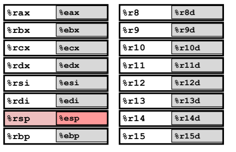
The HW/SW Interface

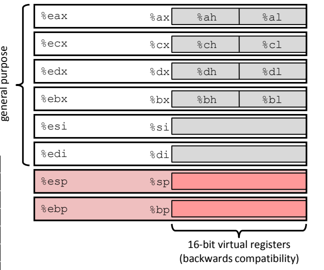
The x86 ISA: Arithmetic and Control



Recap: Machine Programming Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics
  - Registers
  - Operands
  - Move
- Intro to x86-64





Recap: Machine Programming Basics

- Operand Specifiers

Type	Form	Operand value	Name
Immediate	`$Imm`	`Imm`	Immediate
Register	`Ea`	`R[Ea]`	Register
Memory	`Imm`	`M[Imm]`	Absolute
	`(Eb)`	`M[R[Eb]]`	Indirect
	`Imm(Eb)`	`M[R[Eb] + Imm]`	Base + displacement
	`(Eb, Ei)`	`M[R[Eb] + R[Ei]]`	Indexed
	`Imm(Eb, Ei)`	`M[R[Eb] + R[Ei] + Imm]`	Indexed
	`(, Ei, s)`	`M[R[Ei]*s]`	Scaled indexed
	`Imm(, Ei, s)`	`M[R[Ei]*s + Imm]`	Scaled indexed
	`(Eb, Ei, s)`	`M[R[Eb] + R[Ei]*s]`	Scaled indexed
	`Imm(Eb, Ei, s)`	`M[R[Eb] + R[Ei]*s + Imm]`	Scaled indexed

Recap: Machine Programming Basics

- Data Movement Operations

Instruction		Effect	Description
MOV	**S, D**	**D ← S**	**Move**
`movb`		move byte	
`movw`		move word (16-bit)	
`movl`		move double word (32-bit)	
`movq`		move quad word (64-bit)	
MOVS	**S, D**	**D ← SignExtend(S)**	**Move with sign extension**
`movsb[w,l,q]`		move sign-extended byte to word, double word, quad word	
`movsw[l,q]`		move sign-extended word to double word, quad word	
`movslq`		move sign-extended double word to quad word	
MOVZ	**S, D**	**D ← ZeroExtend(S)**	**Move with zero extension**
`movzb[w,l,q]`		move zero-extended byte to word, double word, quad word	
`movzw[l,q]`		move zero-extended word to double word, quad word	
`movzlq`		move zero-extended double word to quad word	
STACK			**Stack operations**
`pushl`	S	`R[%esp] ← R[%esp] - 4;` `M[R[%esp]] ← S`	push double word onto stack
`popl`	D	`D ← M[R[%esp]];` `R[%esp] ← R[%esp] + 4`	pop double word from stack

Machine Programming: Arithmetic & Control

- **Complete addressing mode, address computation (`leal`)**
- Arithmetic operations
- Control: Condition codes
- Conditional branches

Acknowledgement: slides based on the cs:app2e material

Complete Memory Addressing Modes

- General Form:

  `D(Rb,Ri,S)      Mem[ Reg[Rb] + S*Reg[Ri] + D ]`

- D: Constant "displacement" (represented with 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?***)

- Special Cases

```
(Rb,Ri)        Mem[Reg[Rb]+Reg[Ri]]
D(Rb,Ri)       Mem[Reg[Rb]+Reg[Ri]+D]
(Rb,Ri,S)      Mem[Reg[Rb]+S*Reg[Ri]]
```

Address Computation Examples

`%edx`	`0xf000`
`%ecx`	`0x0100`

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

Address Computation Examples

`%edx`	`0xf000`
`%ecx`	`0x0100`

Expression	Address Computation	Address
`0x8(%edx)`	`0xf000 + 0x8`	`0xf008`
`(%edx,%ecx)`	`0xf000 + 0x100`	`0xf100`
`(%edx,%ecx,4)`	`0xf000 + 4*0x100`	`0xf400`
`0x80(,%edx,2)`	`2*0xf000 + 0x80`	`0x1e080`

Address Computation Instruction `leal`

- **`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
```


Machine Programming: Arithmetic & Control

- Complete addressing mode, address computation (`leal`)
- **Arithmetic operations**
- Control: Condition codes
- Conditional branches

Some Arithmetic Operations

- Two Operand Instructions:

Instruction	Effect	Description
`leal S, D`	D ← &S	load effective address
`add S, D`	D ← D + S	add
`sub S, D`	D ← D - S	subtract
`imul S, D`	D ← D * S	multiply
`xor S, D`	D ← D ^ S	exclusive-or
`or S, D`	D ← D | S	or
`and S, D`	D ← D & S	and
`sal k, D`	D ← D << k	left shift
`shl k, D`	D ← D << k	left shift (same as sal)
`sar k, D`	D ← D $>>_A$ k	arithmetic right shift
`shr k, D`	D ← D $>>_L$ k	logical right shift

Note: shift amount (k) given as a (5-bit) immediate or in %cl

  - No distinction between signed and unsigned int

Some Arithmetic Operations

- Single Operand Instructions:

Instruction	Effect	Description
`inc D`	D ← D + 1	increment
`dec D`	D ← D - 1	decrement
`neg D`	D ← -D	negate
`not D`	D ← ~D	complement

Some Arithmetic Operations

- Special Arithmetic Operations

Instruction		Effect	Description
imull	S	R[%edx]:R[%eax] ← S x R[%eax]	signed full multiply
mull	S	R[%edx]:R[%eax] ← S x R[%eax]	unsigned full multiply
cltd	S	R[%edx]:R[%eax] ← SignExtend(R[S])	convert to quad word
idivl	S	R[%edx] ← R[%edx]:R[%eax] mod S; R[%eax] ← R[%edx]:R[%eax] ÷ S	signed divide
divl	S	R[%edx] ← R[%edx]:R[%eax] mod S; R[%eax] ← R[%edx]:R[%eax] ÷ S	unsigned divide

  - R[%edx]:R[%eax] viewed as a single 64-bit quad word

Example: Arithmetic Operations

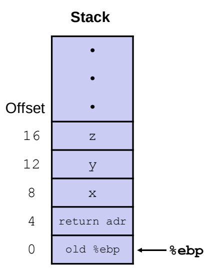
```
int arith
  (int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```

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

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

  popl   %ebp                     } Finish
  ret                             }
```

"abusing" `leal` as an arithmetic operation

Understanding arith

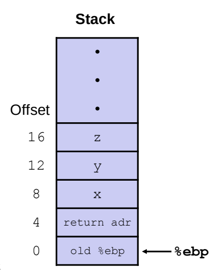
```
int arith(int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```




```
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 compiling
  `(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               } 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           } 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           } 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;
}
```

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

```
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)
```


Machine Programming: Arithmetic & Control

- Complete addressing mode, address computation (`leal`)
- Arithmetic operations
- **Control: Condition codes**
- Conditional branches

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`)

`%eax`	General purpose registers
`%ecx`	General purpose registers
`%edx`	General purpose registers
`%ebx`	General purpose registers
`%esi`	General purpose registers
`%edi`	General purpose registers
`%esp`	Current stack top
`%ebp`	Current stack frame
`%eip`	Instruction pointer
`CF` `ZF` `SF` `OF`	Condition codes

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 IA32 documentation on eTL → Additional Resources

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`

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 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

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

```
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
```


Reading Condition Codes: x86-64

- SetX Instructions:
  - Set single byte based on combination of condition codes
  - Does not alter remaining 3 bytes

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

```
long lgt (long x, long y)
{
  return x > y;
}
```

Body (same for both)

```
xorl %eax, %eax      # eax = 0
cmpq %rsi, %rdi      # compare x and y
setg %al             # al = x > y
```

Is `%rax` zero?
Yes: 32-bit instructions set high order 32 bits to 0!

Machine Programming: Arithmetic & Control

- Complete addressing mode, address computation (`leal`)
- Arithmetic operations
- Control: Condition codes
- **Conditional branches & moves**

Jumping

- jX Instructions
  - Jump to different part of code depending on condition codes

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)

Conditional Branch Example

```
int absdiff(int x, int y)
{
    int result;
    if (x > y) {
      result = x-y;
    } else {
      result = y-x;
    }
    return result;
}
```

```
absdiff:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L6
    subl    %eax, %edx
    movl    %edx, %eax
    jmp .L7
.L6:
    subl %edx, %eax
.L7:
    popl %ebp
    ret
```

setup
evaluate condition
if part
else part
finish

Conditional Branch Example (Cont.)

```
int goto_ad(int x, int y)
{
  int result;
  if (x <= y) goto Else;
  result = x-y;
  goto Exit;
Else:
  result = y-x;
Exit:
  return result;
}
```

- C allows "goto" as means of transferring control
  - Closer to machine-level programming style
- Never use goto in C code!

```
absdiff:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L6
    subl    %eax, %edx
    movl    %edx, %eax
    jmp .L7
.L6:
    subl %edx, %eax
.L7:
    popl %ebp
    ret
```

setup
evaluate condition
if part
else part
finish

Conditional Branch Example (Cont.)

```
int goto_ad(int x, int y)
{
  int result;
  if (x <= y) goto Else;
  result = x-y;
  goto Exit;
Else:
  result = y-x;
Exit:
  return result;
}
```

```
absdiff:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L6
    subl    %eax, %edx
    movl    %edx, %eax
    jmp .L7
.L6:
    subl %edx, %eax
.L7:
    popl %ebp
    ret
```

setup
evaluate condition
if part
else part
finish

Conditional Branch Example (Cont.)

```
int goto_ad(int x, int y)
{
  int result;
  if (x <= y) goto Else;
  result = x-y;
  goto Exit;
Else:
  result = y-x;
Exit:
  return result;
}
```

```
absdiff:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L6
    subl    %eax, %edx
    movl    %edx, %eax
    jmp .L7
.L6:
    subl %edx, %eax
.L7:
    popl %ebp
    ret
```

setup
evaluate condition
if part
else part
finish

Conditional Branch Example (Cont.)

```
int goto_ad(int x, int y)
{
  int result;
  if (x <= y) goto Else;
  result = x-y;
  goto Exit;
Else:
  result = y-x;
Exit:
  return result;
}
```

```
absdiff:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L6
    subl    %eax, %edx
    movl    %edx, %eax
    jmp .L7
.L6:
    subl %edx, %eax
.L7:
    popl %ebp
    ret
```

setup
evaluate condition
if part
else part
finish

General Conditional Expression Translation

C Code

```
val = Test ? Then_Expr : Else_Expr;
```

```
val = x>y ? x-y : y-x;
```

Goto Version

```
  nt = !Test;
  if (nt) goto Else;
  val = Then_Expr;
  goto Done;
Else:
  val = Else_Expr;
Done:
  . . .
```

- Test is expression returning integer
  - = 0 interpreted as false
  - ≠ 0 interpreted as true
- Create separate code regions for then & else expressions
- Execute appropriate one

Using Conditional Moves (`cmovX`)

- Conditional Move Instructions
  - Instruction supports:
    if (Test) Dest ← Src
  - Supported in post-1995 x86 processors
  - GCC does not always use them
    - Wants to preserve compatibility with ancient processors
    - Enabled for x86-64
    - Use switch `-march=686` for IA32
- Why?
  - Branches are very disruptive to instruction flow through pipelines
  - Conditional move do not require control transfer

C Code

```
val = Test
  ? Then_Expr
  : Else_Expr;
```

Goto Version

```
  tval = Then_Expr;
  result = Else_Expr;
  t = Test;
  if (t) result = tval;
  return result;
```


Conditional Move Example: x86-64

```
int absdiff(int x, int y) {
    int result;
    if (x > y) {
        result = x-y;
    } else {
        result = y-x;
    }
    return result;
}
```

```
absdiff:                   # x in %edi, y in %esi
  movl    %edi, %eax       # eax = x
  movl    %esi, %edx       # edx = y
  subl    %esi, %eax       # eax = x-y
  subl    %edi, %edx       # edx = y-x
  cmpl    %esi, %edi       # Compare x:y
  cmovle  %edx, %eax       # eax = edx if <=
  ret
```


Bad Cases for Conditional Move

Expensive Computations

```
val = Test(x) ? Hard1(x) : Hard2(x);
```

- Both values get computed
- Only makes sense when computations are very simple

Risky Computations

```
val = p ? *p : 0;
```

- Both values get computed
- May have undesirable effects

Computations with side effects

```
val = x > 0 ? x*=7 : x+=3;
```

- Both values get computed
- Must be side-effect free

Summary

- Arithmetic & Control
  - Complete addressing mode, address computation (`leal`)
  - Arithmetic operations
  - Control: Condition codes
  - Conditional branches & conditional moves
- Next Lecture
  - Loops (For, While)
  - Switch statements
  - Stack
  - Call / return
  - Procedure call discipline