CS 33: Introduction to Computer Organization

Week 3 TA: Mingda Li

Today:

Lab1

x86 Assembly

x86 Control (if, while) [not switch and function this week]

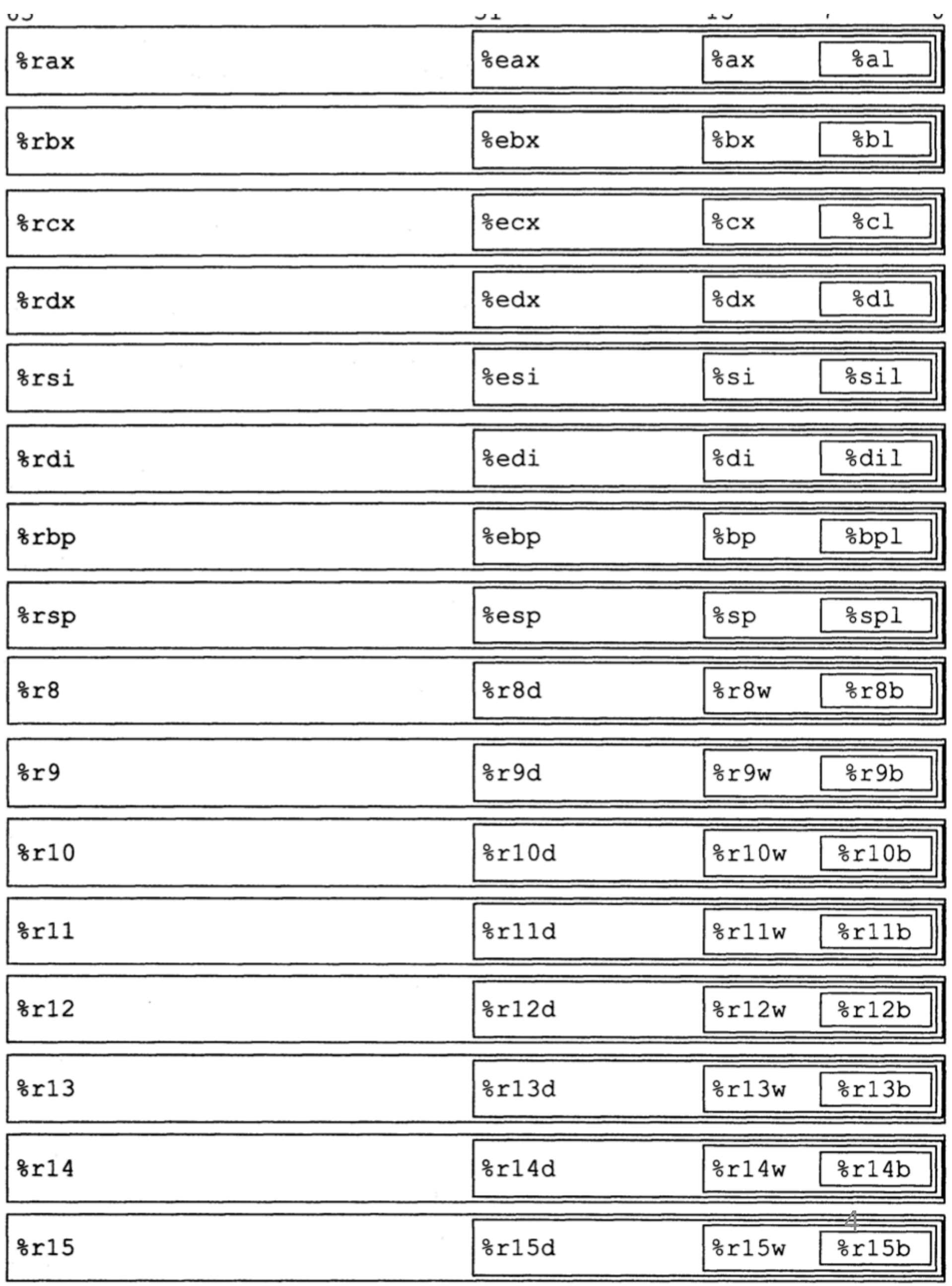
X86 Assembly

- X86 Organization:
- 1. Registers
- Extremely small physical containers that each store a number of bits.

- A 64-bit addressable machine will have registers that are 64-bits.
- In such a case, each register holds 8 bytes
- The access time is extremely quick.

x86 Organization: Registers

- x86-64 contains 16 general purpose registers.
- Rax return value
- Rsp pointer on stack: points to the top of the stack frame (lowest address).



x86 Organization: Registers

- _h: upper 8-bits of lower 16-bits
- _l: lower 8-bits
- _x :lower 16-bits.
- e_x : lower 32-bits (e stands for 'extended').
- r_x : full 64-bit register

General Purpose Registers (A, B, C and D)

<table>
<tr><th>64</th><th>56</th><th>48</th><th>40</th><th>32</th><th>24</th><th>16</th><th>8</th></tr>
<tr><td colspan="8">R?X</td></tr>
<tr><td colspan="4"></td><td colspan="4">E?X</td></tr>
<tr><td colspan="6"></td><td colspan="2">?X</td></tr>
<tr><td colspan="6"></td><td>?H</td><td>?L</td></tr>
</table>

x86 Assembly

- AT&T's (also GNU/GAS) x86 notation (the notation that we will use):
  [op] [src] [dst]

- Intel's x86 notation: [op] [dst] [src]

x86 Assembly

- What does this (GNU/GAS/AT&T) syntax look like?
- [operation] [source] [destination]
- movl $0xdea1, %rax
- addq %rbx %rcx

x86 Assembly

- The mov_ family: move data from the source to the destination.
- movb : move a byte
  movw : move a word (16 bits)
  movl : move a long/double word (32 bits)
  movq : move a quad word (64 bits)
- The suffix determines how much data to move.

x86 Assembly

- x86 Assembly: Addressing Modes Consider:
- movq %rax, (%rbx) <--- ?

- The parentheses () indicate a memory operation.

- That is, the source and destination operands are able to refer to values that are located in memory, rather than just registers.

x86 Assembly

- parentheses () means:

- Treat the bit vector within as a memory address.
- Go follow that address into memory and get the value at that address.

x86 Assembly

- E.g
- %rax = 0xFEEDABBA and %rbx = 0x80.
- movq %rax, %rbx
- Result: %rax = 0xFEEDABBA, %rbx = 0xFEEDABBA

- movq %rax, (%rbx)

- Result: the value that is located in memory address 0x80 is set as 0xFEEDABBA.
- In a more C-like form, this is essentially: MEM[0x80] = 0xFEEDABBA; or *(0x80) = 0xFEEDABBA;
- movq %rax, (%rbx)

x86 Assembly

- The '$' symbol prefix indicates an "immediate" which is constant number value.
- If %rax = 0xb1ab
- movl $0xdea1, %rax
- Result: %rax = 0xdea1

x86 Assembly: Advanced Addressing Modes

- IMM(R1, R2, S) : Scaled and displaced array access.
  - Intended usage:
  - R1 : Base array address
  - R2 : Index into array
  - S : Size of array data type in bytes
  - IMM : Displacement
- movq A(B, C, D), %rax
  - %rax = *(A + B + C*D)

x86 Assembly: Advanced Addressing Modes

D(R1) : Base + displacement addressing

- If %rax = 0x10.
- movq 8(%rax), %rbx
  - Result: %rbx gets the value at memory address 0x10 + 8 = 0x18, *not* the number 0x18
  - %rbx = *(%rax + 8).

x86 Assembly: Advanced Addressing Modes

- IMM(R1, R2, S) : Scaled and displaced array access.
  - If %rax = 0x400, %rbx = 2, S = 2, and D = 20.
  - movl 0x20(%rax, %rbx, 2), %rcx
    - Result: The value at memory address 0x400 + 0x2*2 + 0x20 is placed in %rcx.

x86 Assembly: lea_

- lea_ : “load effective address”, Compare to movq
- movq (%rax), %rbx => The value at memory address contained by %rax is moved to %rbx

- leaq (%rax), %rbx => The value in %rax itself is moved to %rbx.

- movq 4(%rax,%rbx, 2), %rcx.
- This means %rcx = MEM[%rax + %rbx * 2 + 4] leaq 4(%rax,%rbx, 2), %rcx.

- leaq 4(%rax,%rbx, 2), %rcx.
- This means %rcx = %rax + %rbx * 2 + 4

Other instructions

`addq`	*Src,Dest*	Dest = Dest + Src
`subq`	*Src,Dest*	Dest = Dest – Src
`imulq`	*Src,Dest*	Dest = Dest * Src
`salq`	*Src,Dest*	Dest = Dest << Src
`sarq`	*Src,Dest*	Dest = Dest >> Src
`shrq`	*Src,Dest*	Dest = Dest >> Src
`xorq`	*Src,Dest*	Dest = Dest ^ Src
`andq`	*Src,Dest*	Dest = Dest & Src
`orq`	*Src,Dest*	Dest = Dest | Src

Control Flag

- Motivation: extract some information about certain values
  - Example: something you do in the data lab
- Single Bit Registers （explain in next slide)
  - CF: Carry Flag
  - SF: Sign Flag
  - ZF: Zero Flag
  - OF: Overflow Flag
- Things can Influence the control flags
  - Implicitly Set By Arithmetic Operations
    e.g. addition of 2’s complement
  - Explicitly Set by Compare Instruction
    e.g. comparison

- Carry Flag (CF)
  - CF = 1 if the most recent operation caused the most significant bits to have a carry out. Otherwise, CF = 0.
  - Informal usage: the purpose of this is to check for unsigned overflow.
  - If t = a+b, then CF = 1 if
    - (unsigned) t < (unsigned) a
  - Set by most arithmetic instructions and some

- Zero Flag (ZF)
  - ZF = 1 if the result of the most recent operation is zero. Otherwise, ZF = 0.
  - Informal usage: check if two values are equal, check if an operation resulted in a zero.
- Sign Flag (SF)
  - SF = 1 if the result of the operation has the most significant bit as 1. Otherwise SF = 0.
  - This sets the flag if the number is negative
  - If t = a+b, then SF = 1 if:
    - $t < 0$
  - Set by arithmetic (except for mul and div), boolean/bitwise, cmp, and test.

- Overflow Flag (OF)
  - If t = a+b, then SF = 1 if:
    - (a < 0) == (b < 0) && (t < 0) != (a < 0)
  - OF is set if the above expression is 1.
  - Informal usage: This effectively checks for signed overflow.

- The Carry Flag detects unsigned overflow and the Overflow Flag detects signed overflow.

Control Flag

- cmp S2, S1 : Sets the flags based on S1 – S2, but doesn't change S1.
- test S2, S1 : Sets the flags based on S1&S2 but does not change S1.

- After setting the flags, you can use them in two ways:
- 1.Manually set a register based on the state of the flags.
- 2.Use a conditional instruction which does something based on the state of the flags.

Use conditional operations:

- Conditional move (cmov_)
- cmove S, D : move only if ZF is 1.
- cmovs S, D : D = S, but only is SF is 1.

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

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

```
absdiff:
    pushl   %ebp                 } Setup
    movl    %esp, %ebp
    movl    8(%ebp), %edx        <- X to edx
    movl    12(%ebp), %eax       <- Y to eax
    cmpl    %eax, %edx           } Body1
    jle     .L7
    subl    %eax, %edx
    movl    %edx, %eax
.L8:
    leave                        } Finish
    ret
.L7:
    subl    %edx, %eax           } Body2
    jmp     .L8
```

X to edx
Y to eax

the return value is placed into %rax.

Do-While Loop Example

C Code

```
long pcount_do
  (unsigned long x) {
  long result = 0;
  do {
    result += x & 0x1;
    x >>= 1;
  } while (x);
  return result;
}
```

Goto Version

```
long pcount_goto
  (unsigned long x) {
  long result = 0;
 loop:
  result += x & 0x1;
  x >>= 1;
  if(x) goto loop;
  return result;
}
```

- Count number of 1’s in argument x(“popcount”)
- Use conditional branch to either continue looping or to exit

Do-While Loop Example

Goto Version

```
long pcount_goto
  (unsigned long x) {
  long result = 0;
 loop:
  result += x & 0x1;
  x >>= 1;
  if(x) goto loop;
  return result;
}
```

Register	Use(s)
%rdi	Argument **x**
%rax	**result**

```
    movl    $0, %eax     #  result = 0
 .L2:                  # loop:
    movq    %rdi, %rdx
    andl    $1, %edx     #  t = x & 0x1
    addq    %rdx, %rax   #  result += t
    shrq    %rdi         #  x >>= 1
    jne     .L2          #  if (x) goto loop
    rep; ret
```