CSE 351
Section 4

Download the Handout!

https://courses.cs.washington.edu/courses/cse351/20su/sections/04/cse351_sec4.pdf

Solutions will be posted this evening.

x86-64 Assembly

Midterm Reference Sheet

You may not be taking a midterm for this class, but the reference sheet is still a great resource for x86-64 in particular.

You can find it on the website here:
https://courses.cs.washington.edu/courses/cse351/20su/exams/ref-mt.pdf

CSE 351 Reference Sheet (Midterm)

Binary	Decimal	Hex
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}
1	2	4	8	16	32	64	128	256	512	1024

IEEE 754 FLOATING-POINT STANDARD

Value: $\pm 1 \times \text{Mantissa} \times 2^{\text{Exponent}}$

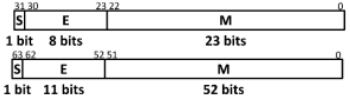
Bit fields: $(-1)^S \times 1.M \times 2^{(E-\text{bias})}$

where Single Precision Bias = 127,
Double Precision Bias = 1023.

IEEE 754 Symbols

E	M	Meaning
all zeros	all zeros	± 0
all zeros	non-zero	± denorm num
1 to MAX-1	anything	± norm num
all ones	all zeros	± ∞
all ones	non-zero	NaN

IEEE Single Precision and Double Precision Formats:



Assembly Instructions

`mov a, b`	Copy from `a` to `b`.
`movs a, b`	Copy from `a` to `b` with sign extension. Needs *two* width specifiers.
`movz a, b`	Copy from `a` to `b` with zero extension. Needs *two* width specifiers.
`lea a, b`	Compute address and store in `b`. *Note:* the scaling parameter of memory operands can only be 1, 2, 4, or 8.
`push src`	Push `src` onto the stack and decrement stack pointer.
`pop dst`	Pop from the stack into `dst` and increment stack pointer.
`call <func>`	Push return address onto stack and jump to a procedure.
`ret`	Pop return address and jump there.
`add a, b`	Add from `a` to `b` and store in `b` (and sets flags).
`sub a, b`	Subtract `a` from `b` (compute `b-a`) and store in `b` (and sets flags).
`imul a, b`	Multiply `a` and `b` and store in `b` (and sets flags).
`and a, b`	Bitwise AND of `a` and `b`, store in `b` (and sets flags).
`sar a, b`	Shift value of `b` *right* (*arithmetic*) by `a` bits, store in `b` (and sets flags).
`shr a, b`	Shift value of `b` *right* (*logical*) by `a` bits, store in `b` (and sets flags).
`shl a, b`	Shift value of `b` *left* by `a` bits, store in `b` (and sets flags).
`cmp a, b`	Compare `b` with `a` (compute `b-a` and set condition codes based on result).
`test a, b`	Bitwise AND of `a` and `b` and set condition codes based on result.
`jmp <label>`	Unconditional jump to address.
`j* <label>`	Conditional jump based on condition codes (*more on next page*).
`set* a`	Set byte `a` to 0 or 1 based on condition codes.

Conditionals

Instruction		`(op) s, d`	`test a, b`	`cmp a, b`
`je`	"Equal"	`d (op) s == 0`	`b & a == 0`	`b == a`
`jne`	"Not equal"	`d (op) s != 0`	`b & a != 0`	`b != a`
`js`	"Sign" (negative)	`d (op) s < 0`	`b & a < 0`	`b-a < 0`
`jns`	(non-negative)	`d (op) s >= 0`	`b & a >= 0`	`b-a >= 0`
`jg`	"Greater"	`d (op) s > 0`	`b & a > 0`	`b > a`
`jge`	"Greater or equal"	`d (op) s >= 0`	`b & a >= 0`	`b >= a`
`jl`	"Less"	`d (op) s < 0`	`b & a < 0`	`b < a`
`jle`	"Less or equal"	`d (op) s <= 0`	`b & a <= 0`	`b <= a`
`ja`	"Above" (unsigned >)	`d (op) s > 0U`	`b & a > 0U`	`b >U a`
`jb`	"Below" (unsigned <)	`d (op) s < 0U`	`b & a < 0U`	`b <U a`

Registers

Name	Convention	Name of "virtual" register		
		Lowest 4 bytes	Lowest 2 bytes	Lowest byte
`%rax`	Return value – **Caller** saved	`%eax`	`%ax`	`%al`
`%rbx`	**Callee** saved	`%ebx`	`%bx`	`%bl`
`%rcx`	Argument #4 – **Caller** saved	`%ecx`	`%cx`	`%cl`
`%rdx`	Argument #3 – **Caller** saved	`%edx`	`%dx`	`%dl`
`%rsi`	Argument #2 – **Caller** saved	`%esi`	`%si`	`%sil`
`%rdi`	Argument #1 – **Caller** saved	`%edi`	`%di`	`%dil`
`%rsp`	Stack Pointer	`%esp`	`%sp`	`%spl`
`%rbp`	**Callee** saved	`%ebp`	`%bp`	`%bpl`
`%r8`	Argument #5 – **Caller** saved	`%r8d`	`%r8w`	`%r8b`
`%r9`	Argument #6 – **Caller** saved	`%r9d`	`%r9w`	`%r9b`
`%r10`	**Caller** saved	`%r10d`	`%r10w`	`%r10b`
`%r11`	**Caller** saved	`%r11d`	`%r11w`	`%r11b`
`%r12`	**Callee** saved	`%r12d`	`%r12w`	`%r12b`
`%r13`	**Callee** saved	`%r13d`	`%r13w`	`%r13b`
`%r14`	**Callee** saved	`%r14d`	`%r14w`	`%r14b`
`%r15`	**Callee** saved	`%r15d`	`%r15w`	`%r15b`

Sizes

C type	x86-64 suffix	Size (bytes)
`char`	`b`	1
`short`	`w`	2
`int`	`l`	4
`long`	`q`	8

Address Computation

We can do more complicated memory accesses like so:

- `D(Rb, Ri, S)`
  - `Rb` - base register
  - `Ri` - index register
  - `S` - scale factor (1, 2, 4, 8)
  - `D` - displacement
  - Result is `D + Rb + Ri * S`

So `0x400(%rdi, %rsi, 4)` evaluates to `%rdi + 4 * %rsi + 0x400`. This is very useful for accessing elements in an array, and also for use in conjunction with `lea` (which does this address computation, but stores the raw result instead of accessing memory at the computed address).

Condition Codes

Condition codes include the zero, sign, carry (unsigned overflow), and (signed) overflow flags. They are stored on the processor in their own register.

- They are implicitly set by arithmetic operations:
  - `addq src, dst`
    - `r = dst + src` (result used to set flags)
- There are also instructions to set only the condition codes:
  - `cmp a, b`
    - `r = b - a` (result sets flags, but is not stored)
  - `test a, b`
    - `r = a & b` (result sets flags, but is not stored)

Control Flow

The condition codes are often used in combination with j* (jump) and set* instructions. These instructions take one operand and change the instruction pointer or given byte respectively depending on different combinations of the condition codes.

Conditionals

Instruction		(op) s, d	test a, b	cmp a, b
je	"Equal"	d (op) s == 0	b & a == 0	b == a
jne	"Not equal"	d (op) s != 0	b & a != 0	b != a
js	"Sign" (negative)	d (op) s < 0	b & a < 0	b-a < 0
jns	(non-negative)	d (op) s >= 0	b & a >= 0	b-a >= 0
jg	"Greater"	d (op) s > 0	b & a > 0	b > a
jge	"Greater or equal"	d (op) s >= 0	b & a >= 0	b >= a
jl	"Less"	d (op) s < 0	b & a < 0	b < a
jle	"Less or equal"	d (op) s <= 0	b & a <= 0	b <= a
ja	"Above" (unsigned >)	d (op) s > 0U	b & a > 0U	$b >_U a$
jb	"Below" (unsigned <)	d (op) s < 0U	b & a < 0U	$b <_U a$

Procedure Basics

There are several instructions which manipulate the stack. Note that `%rsp` is the stack pointer, which holds the address of the last thing pushed to the stack.

- `push x`
  - Decrements stack pointer (stack grows down!) and places value x at %rsp
- `pop x`
  - Copies value at %rsp to x and increments stack pointer.
- `call x`
  - Pushes address of following instruction (“return address”) to stack and jumps to x.
- `ret`
  - Pops to %rip (gets return address from stack and sets instruction pointer)

Passing Arguments

By convention, **`%rax`** is used for the return value.

The first six arguments to procedure calls are passed in the order:

`%rdi`, **`%rsi`**, **`%rdx`**, **`%rcx`**, **`%r8`**, **`%r9`**

(Diane's silk dress cost $89)

- What if we have more than 6 arguments?
- What if need to preserve a register's value before and after a function call?

Exercises

Exercise 1

Convert the following C function into x86-64 assembly code. You are not being judged on the efficiency of your code – just correctness.

```
long happy(long *x, long y, long z) {
    if (y > z)
        return z + y;
    else
                return *x;
}
```

Exercise 1

```
long happy(long *x, long y, long z) {
    if (y > z)
        return z + y;
    else
                return *x;
}
```

Exercise 1

```
long happy(long *x, long y, long z) {
    if (y > z)
        return z + y;
    else
                return *x;
}
```

```
.happy:


.else:
```

Exercise 1

```
long happy(long *x, long y, long z) {
    if (y > z)
        return z + y;
    else
                return *x;
}
```

```
.happy:
    cmpq %rdx, %rsi
    jle  .else

.else:
```

Conditionals

Instruction		(op) s, d	test a, b	cmp a, b
je	"Equal"	d (op) s == 0	b & a == 0	b == a
jne	"Not equal"	d (op) s != 0	b & a != 0	b != a
js	"Sign" (negative)	d (op) s < 0	b & a < 0	b-a < 0
jns	(non-negative)	d (op) s >= 0	b & a >= 0	b-a >= 0
jg	"Greater"	d (op) s > 0	b & a > 0	b > a
jge	"Greater or equal"	d (op) s >= 0	b & a >= 0	b >= a
jl	"Less"	d (op) s < 0	b & a < 0	b < a
jle	"Less or equal"	d (op) s <= 0	b & a <= 0	b <= a
ja	"Above" (unsigned >)	d (op) s > 0U	b & a > 0U	b $>_U$ a
jb	"Below" (unsigned <)	d (op) s < 0U	b & a < 0U	b $<_U$ a

Exercise 1

```
long happy(long *x, long y, long z) {
    if (y > z)
        return z + y;
    else
                return *x;
}
```

```
.happy:
    cmpq %rdx, %rsi
    jle  .else
    leaq (%rsi, %rdx) %rax
    ret

.else:
```

```
addq %rsi, %rdx
movq %rdx, %rax
```

Exercise 1

```
long happy(long *x, long y, long z) {
    if (y > z)
        return z + y;
    else
                return *x;
}
```

```
.happy:
    cmpq %rdx, %rsi
    jle  .else
    leaq (%rsi, %rdx) %rax
    ret

.else:
    movq (%rdi), %rax
    ret
```

Exercise 2

Write an equivalent C function for the following x86-64 code:

```
mystery:
  testl   %edx, %edx
  js      .L3
  cmpl    %esi, %edx
  jge     .L3
  movslq  %edx, %rdx
  movl    (%rdi,%rdx,4), %eax
  ret
.L3:
  movl    $0, %eax
  ret
```

Exercise 2

```
mystery:
  testl   %edx, %edx
  js      .L3
  cmpl    %esi, %edx
  jge     .L3
  movslq  %edx, %rdx
  movl    (%rdi,%rdx,4), %eax
  ret
.L3:
  movl    $0, %eax
  ret
```

Exercise 2

```
mystery:
  testl   %edx, %edx
  js      .L3
  cmpl    %esi, %edx
  jge     .L3
  movslq  %edx, %rdx
  movl    (%rdi,%rdx,4), %eax
  ret
.L3:
  movl    $0, %eax
  ret
```

```
int mystery(? x, int y, int z)
```

Exercise 2

```
mystery:
  testl   %edx, %edx
  js      .L3
  cmpl    %esi, %edx
  jge     .L3
  movslq  %edx, %rdx
  movl    (%rdi,%rdx,4), %eax
  ret
.L3:
  movl    $0, %eax
  ret
```

```
int mystery(? x, int y, int z) {
  if ( )

  else

}
```

Exercise 2

```
mystery:
  testl   %edx, %edx
  js      .L3
  cmpl    %esi, %edx
  jge     .L3
  movslq  %edx, %rdx
  movl    (%rdi,%rdx,4), %eax
  ret
.L3:
  movl    $0, %eax
  ret
```

```
int mystery(? x, int y, int z) {
  if (z >= 0 && z < y)

  else

}
```

Conditionals

Instruction		(op) s, d	test a, b	cmp a, b
je	"Equal"	d (op) s == 0	b & a == 0	b == a
jne	"Not equal"	d (op) s != 0	b & a != 0	b != a
js	"Sign" (negative)	d (op) s < 0	b & a < 0	b-a < 0
jns	(non-negative)	d (op) s >= 0	b & a >= 0	b-a >= 0
jg	"Greater"	d (op) s > 0	b & a > 0	b > a
jge	"Greater or equal"	d (op) s >= 0	b & a >= 0	b >= a
jl	"Less"	d (op) s < 0	b & a < 0	b < a
jle	"Less or equal"	d (op) s <= 0	b & a <= 0	b <= a
ja	"Above" (unsigned >)	d (op) s > 0U	b & a > 0U	b $>_U$ a
jb	"Below" (unsigned <)	d (op) s < 0U	b & a < 0U	b $<_U$ a

Exercise 2

```
mystery:
  testl   %edx, %edx
  js      .L3
  cmpl    %esi, %edx
  jge     .L3
  movslq  %edx, %rdx
  movl    (%rdi,%rdx,4), %eax
  ret
.L3:
  movl    $0, %eax
  ret
```

```
int mystery(int *x, int y, int z) {
  if (z >= 0 && z < y)
        return x[z];

  else

}
```

Exercise 2

```
mystery:
  testl   %edx, %edx
  js      .L3
  cmpl    %esi, %edx
  jge     .L3
  movslq  %edx, %rdx
  movl    (%rdi,%rdx,4), %eax
  ret
.L3:
  movl    $0, %eax
  ret
```

```
int mystery(int *x, int y, int z) {
  if (z >= 0 && z < y)
        return x[z];

  else
        return 0;
}
```