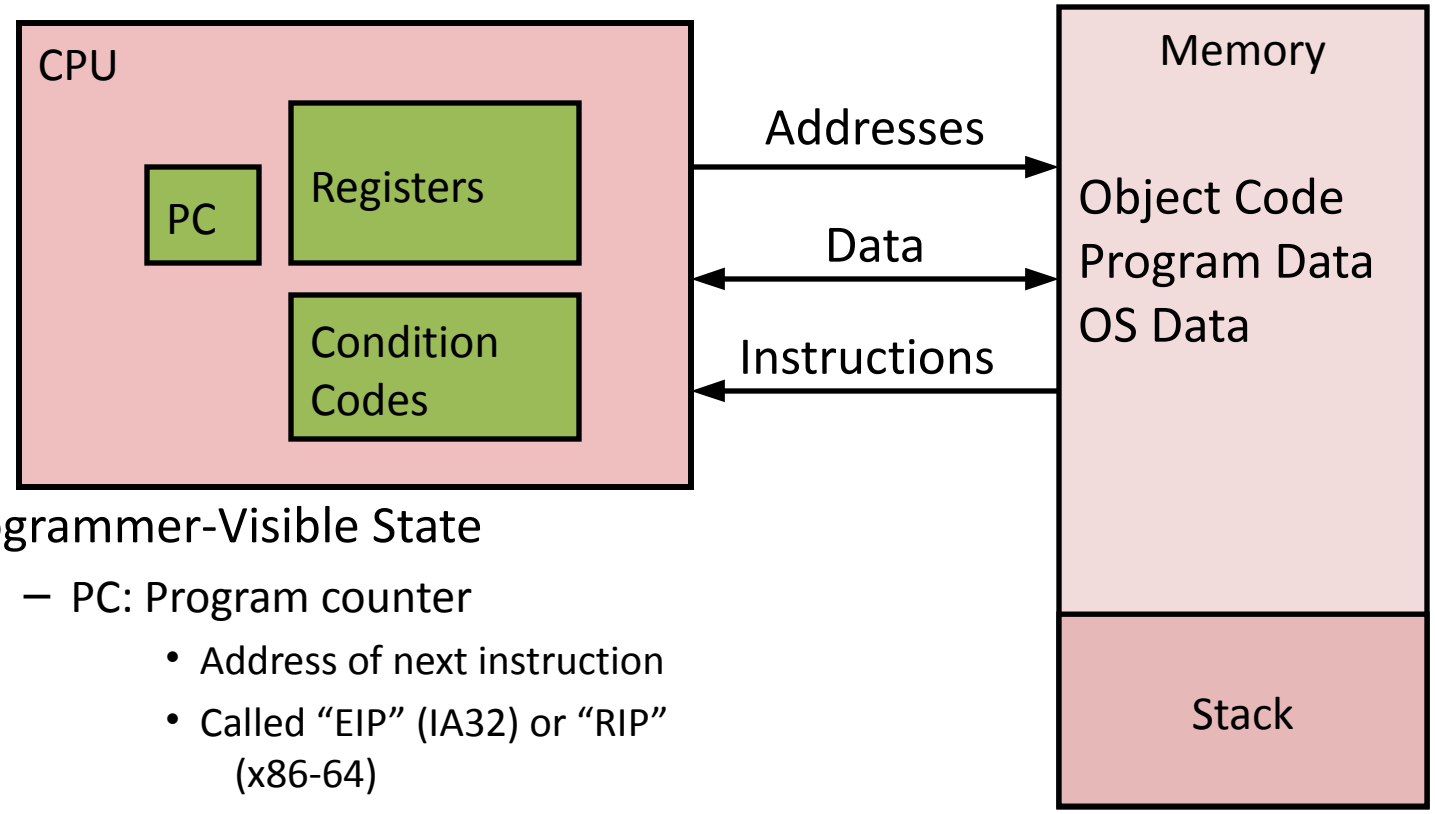
CSE 351
Section 5

2/2/12

Agenda

- C to Assembly and back

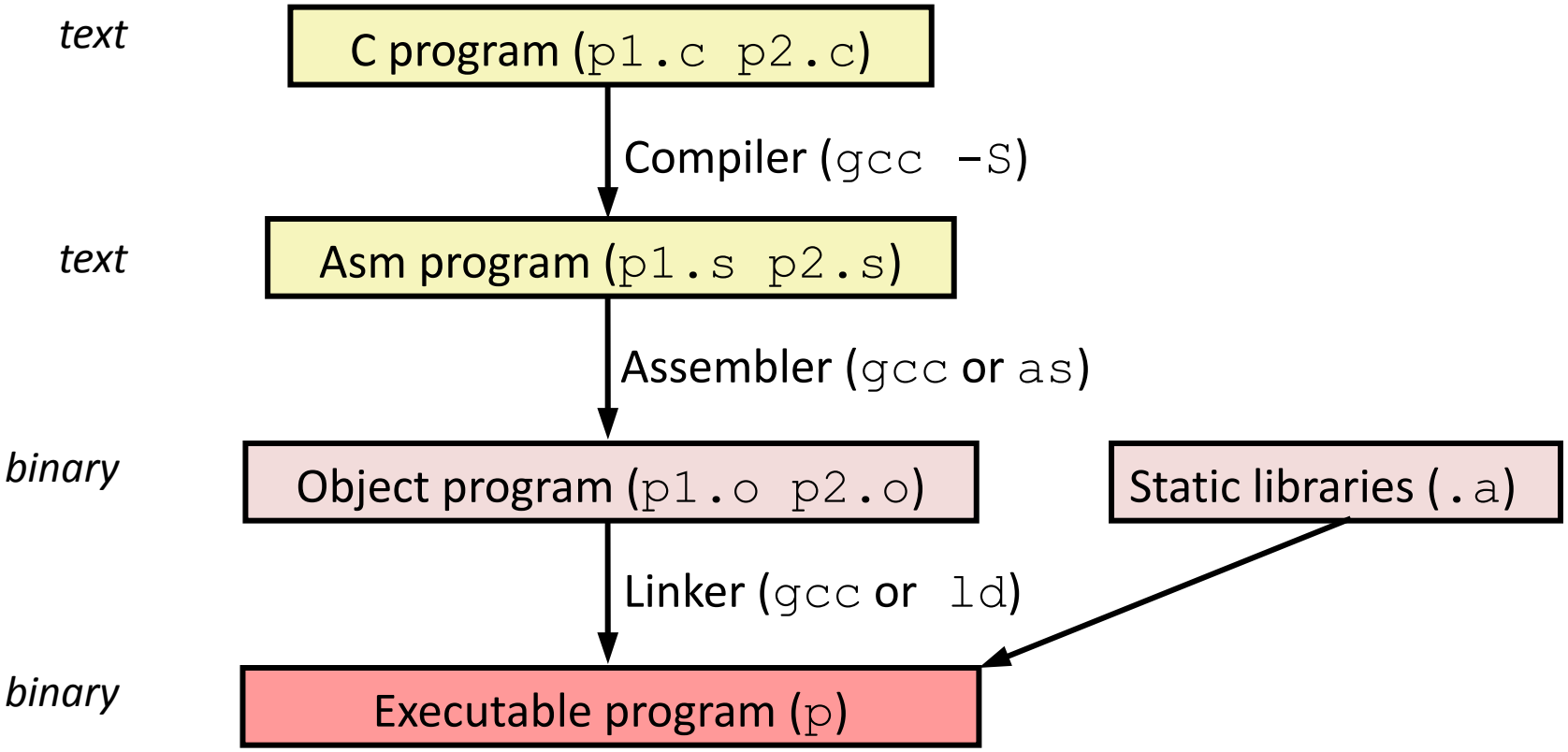
Assembly Programmer's View



- Programmer-Visible State
  - PC: Program counter
    - Address of next instruction
    - Called "EIP" (IA32) or "RIP" (x86-64)
  - Register file
    - Heavily used program data
  - Condition codes
    - Store status information about most recent arithmetic operation
    - Used for conditional branching
  - **Memory**
    - Byte addressable array
    - Code, user data, (some) OS data
    - Includes stack used to support procedures (we'll come back to that)

Turning C into Object Code

- Code in files **`p1.c p2.c`**
- Compile with command: **`gcc -O p1.c p2.c -o p`**
  - Use optimizations (**`-O`**)
  - Put resulting binary in file **`p`**



Compiling Into Assembly (32-bit Example)

C Code

```
int sum(int x, int y)
{
  int t = x+y;
  return t;
}
```

Generated IA32 Assembly

```
sum:
    pushl %ebp
    movl %esp,%ebp
    movl 12(%ebp),%eax
    addl 8(%ebp),%eax
    movl %ebp,%esp
    popl %ebp
    ret
```

Obtain with command

```
gcc -O -S code.c
```

Produces file `code.s`

Compiling Into Assembly (64-bit Example)

C Code

```
int sum(int x, int y)
{
  int t = x+y;
  return t;
}
```

Obtain with command

```
gcc -S code.c
```

Produces file `code.s`

Generated x64 Assembly

```
sum:
    pushq %rbp
    movq %rsp,%rbp
    movl %edi, -20(%rbp)
    movl %esi, -24(%rbp)
    movl -24(%rbp), %eax
    movl -20(%rbp), %edx
    addl %edx, %eax
    movl %eax, -4(%rbp)
    movl -4(%rbp), %eax
    popq %rbp
    ret
```

Compiling Into Assembly (64-bit Example)

C Code

```
int sum(int x, int y)
{
  int t = x+y;
  return t;
}
```

Generated x64 Assembly

```
sum:
    leal (%rdi,%rsi), %eax
    ret
```

Obtain with command

```
gcc -O -S code.c
```

Produces file `code.s`

Three Kinds of Instructions

- Perform arithmetic function on register or memory data

- Transfer data between memory and register
  - Load data from memory into register
  - Store register data into memory

- Transfer control (control flow)
  - Unconditional jumps to/from procedures
  - Conditional branches

Object Code (32-bit Example)

Code for `sum`

```
0x401040 <sum>:
0x55
    0x89
    0xe5
    0x8b
    0x45
    0x0c
    0x03
    0x45
    0x08
    0x89
    0xec
    0x5d
    0xc3
```

- Total of 13 bytes
- Each instruction 1, 2, or 3 bytes
- Starts at address `0x401040`

- Assembler
  - Translates `.s` into `.o`
  - Binary encoding of each instruction
  - Nearly-complete image of executable code
  - Missing linkages between code in different files
- Linker
  - Resolves references between files
  - Combines with static run-time libraries
    - E.g., code for `malloc`, `printf`
  - Some libraries are *dynamically linked*
    - Linking occurs when program begins execution

Example (32-bit)

```
int t = x+y;
```

```
addl 8(%ebp),%eax
```

Similar to expression:

```
x += y
```

More precisely:

```
int eax;
int *ebp;
eax += ebp[2]
```

```
0x401046: 03 45 08
```

- C Code
  - Add two signed integers
- Assembly
  - Add 2 4-byte integers
    - "Long" words in GCC speak
    - Same instruction whether signed or unsigned
  - Operands:
    - **x:** Register **%eax**
    - **y:** Memory **M[%ebp+8]**
    - **t:** Register **%eax**
      - Return function value in **%eax**
- Object Code
  - 3-byte instruction
  - Stored at address **0x401046**

Disassembling Object Code (32-bit)

Disassembled

```
00401040 <_sum>:
   0:      55                   push   %ebp
   1:      89 e5                mov    %esp,%ebp
   3:      8b 45 0c             mov    0xc(%ebp),%eax
   6:      03 45 08             add    0x8(%ebp),%eax
   9:      89 ec                mov    %ebp,%esp
   b:      5d                   pop    %ebp
   c:      c3                   ret
   d:      8d 76 00             lea    0x0(%esi),%esi
```

- Disassembler

  `objdump -d p`

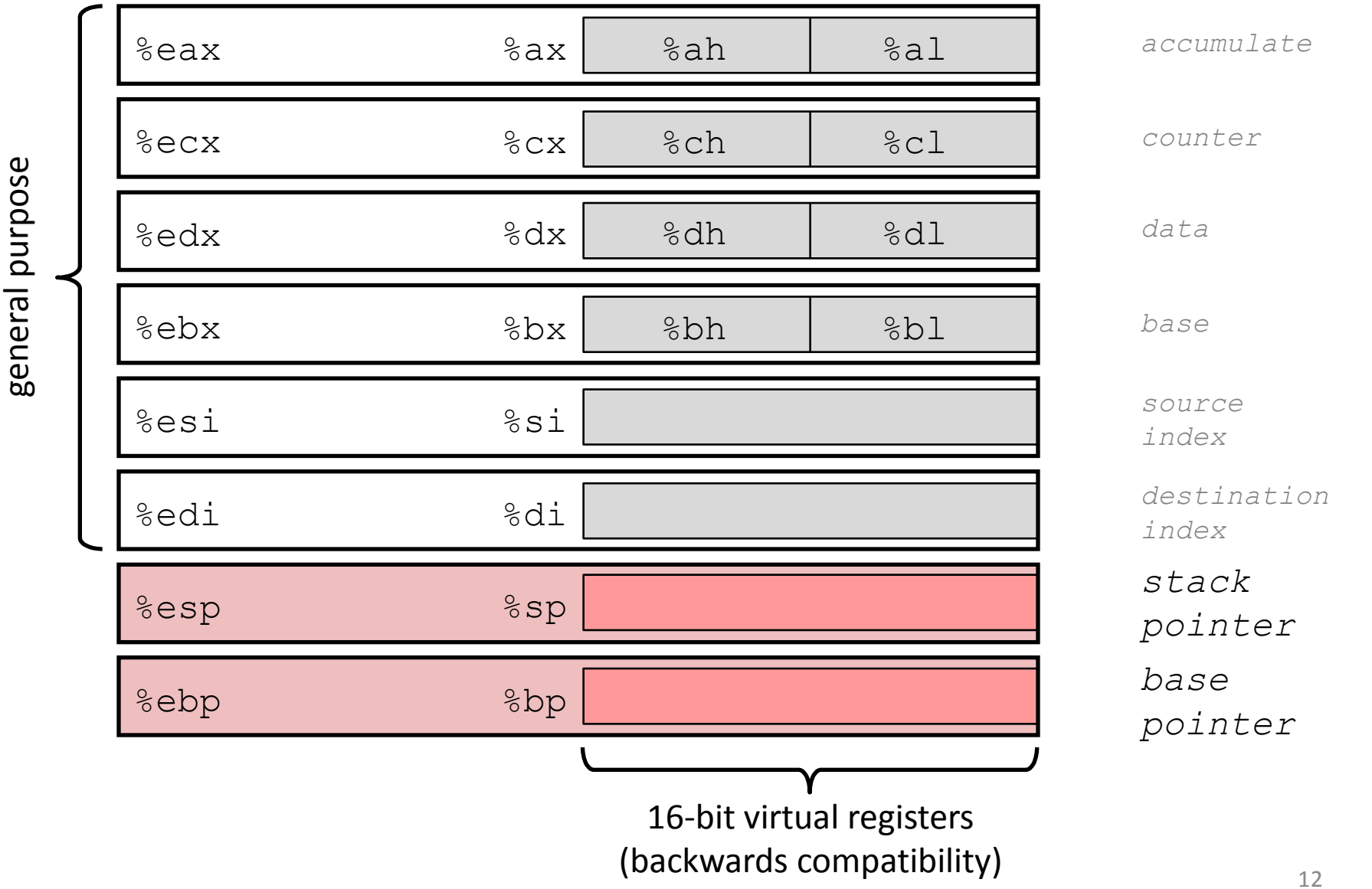
  - Useful tool for examining object code
  - Analyzes bit pattern of series of instructions
  - Produces approximate rendition of assembly code
  - Can be run on either `a.out` (complete executable) or `.o` file

- Can also use GDB

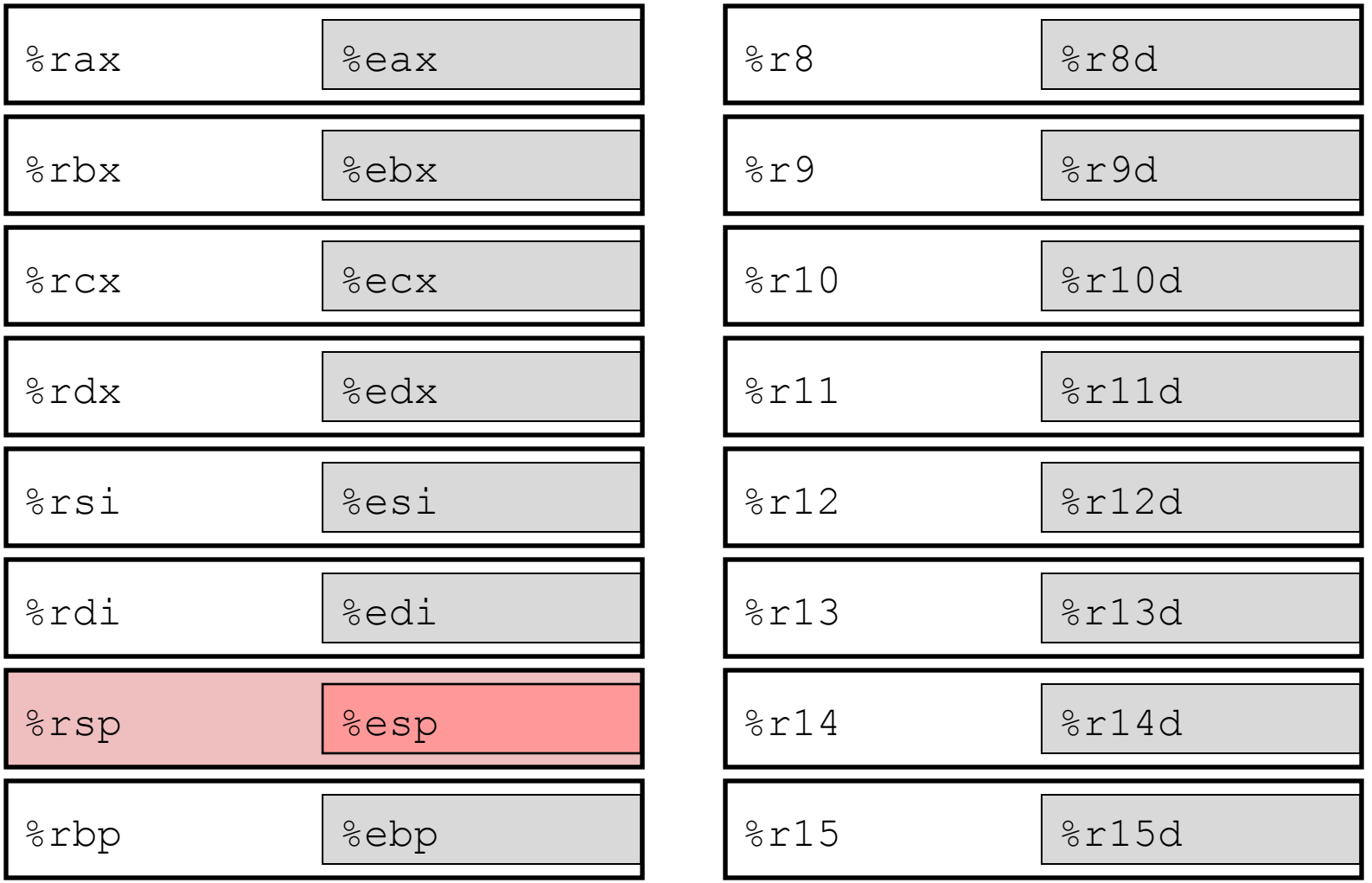
  `$ gdb foo`

  `> disas main`

Integer Registers (IA32)



x86-64 Integer Registers



- Twice the number of registers
- Accessible as 8, 16, 32, 64 bits

x86-64 Integer Registers: Usage Conventions

%rax	Return value
%rbx	Callee saved
%rcx	Argument #4
%rdx	Argument #3
%rsi	Argument #2
%rdi	Argument #1
%rsp	Stack pointer
%rbp	Callee saved

%r8	Argument #5
%r9	Argument #6
%r10	Caller saved
%r11	Caller Saved
%r12	Callee saved
%r13	Callee saved
%r14	Callee saved
%r15	Callee saved