Outline

- PC organization
- x86 instruction set
- gcc inline assembly and calling conventions
PC organization

- One or more CPUs, memory, and device controllers connected through system bus
CPU: “brain”

- Use 16-bit 8086 (1978) as example

- CPU runs instructions
  - while (fetch next instruction)
  - run instruction;

- Needs work space: registers
  - Four 16-bit data registers: AX, BX, CX, DX
  - Each has two 8-bit halves: e.g., AH, AL
  - Very fast, very few

- More work space: memory
  - Array of data cells
  - CPU sends out address on address lines
  - Data comes back on data lines or is written to data lines
Address registers

- **Needs pointers to memory: address registers**
  - **SP**: stack pointer
  - **BP**: frame base pointer
  - **SI**: source index
  - **DI**: destination index

- **Instructions are in memory too!**
  - **IP**: instruction pointer
  - Increment after running each instruction
  - *Can be modified by CALL, RET, JMP, conditional jumps*
Segment registers

- More than $2^{16}$ bytes of physical memory?
  - 8086 has 20-bit addresses $\Rightarrow$ 1 MB RAM

- Segment registers
  - CS: code segment, for fetches via IP
  - SS: stack segment, for load/store via SP and BP
  - DS: data segment, for load/store via other registers
  - ES: another data segment, destination for string operations

- 20 bit address = seg * 16 + 16 bit address
FLAGS

- Want conditional jumps
  - FLAGS - various condition codes of last instruction
    - ZF: zero flag
    - SF: signed flag
    - OF: overflow flag
    - CF: carry flag
    - PF: parity flag
    - IF: interrupt flag, whether interrupts are enabled

- J[N]Z  J[N]S  J[N]O ...
Intel 80386 and AMD K8

- 16-bit addresses and data were painfully small

- 80386 added support for 32 bit (1985)
  - Registers are 32 bits wide
    - E.g., EAX instead of AX

- AMD K8 added support for 64 bit (2003)
  - Codename Althon 64
  - Registers are 64 bits wide
  - RAX instead of EAX
  - x86-64, x64, amd64, intel64: all same thing
Outline

- PC organization
- x86 instruction set
- gcc inline assembly and calling conventions
Syntax

- Intel manual: \texttt{op dst, src}
- AT&T (gcc/gas): \texttt{op src, dst}
  - \texttt{op} uses suffix \texttt{b, w, l} for 8, 16, 32-bit operands

Operands are registers, constants, memory via register, memory via constant

Examples
- \texttt{movl \%ebx, \%edx ; edx = ebx} register
- \texttt{movl $0x123, \%edx ; edx = 0x123} immediate
- \texttt{movl 0x123, \%edx ; edx = *(int32_t*)0x123} direct
- \texttt{movl (%ebx), \%edx ; edx = *(int32_t*)ebx} indirect
- \texttt{movl 4(%ebx), \%edx ; edx = *(int32_t*)(ebx+4)} displaced
Instruction classes

- Data movement: MOV, PUSH, POP, ...
- Arithmetic: TEST, SHL, ADD, AND, ...
- I/O: IN, OUT, ...
- Control: JMP, JZ, JNZ, CALL, RET
- String: MOVSB, REP, ...
- System: INT, IRET
Outline

- PC organization
- x86 instruction set
- gcc inline assembly and calling conventions
Build process: C code to x86 instructions

1. **C code (ASCII text)**
   - \(\rightarrow\) proprocessor
   - proprocessor \(\rightarrow\) **C code with #include etc expanded (ASCII text)**

2. **Assembly code (ASCII text)**
   - \(\rightarrow\) **Assembly code (ASCII text)**
   - **Assembly code (ASCII text)** \(\rightarrow\) **Program image (binary)**
   - **Program image (binary)** \(\rightarrow\) loader

3. **.o (binary, machine readable)**
   - \(\rightarrow\) linker
   - linker \(\rightarrow\) Program image (binary)

gcc inline assembly

- Embed assembly code in C code

- Syntax: `asm ("assembly code")`
  
  e.g., `asm ("movl %eax %ebx")`

- Advanced syntax

```c
asm ( assembler template 
    : output operands /* optional */ 
    : input operands /* optional */ 
    : list of clobbered registers /* optional */ );
```
gcc inline assembly example

```
int a=10, b;
asm ("movl %1, %%eax;
     movl %%eax, %0;"
  :"=r"(b) /* output operands */
  :"r"(a)  /* input operands */
  :"%eax" /* clobbered registers */ );
```

- Equivalent to $b = a$
- Operand number: $%0, %1, \ldots, %n-1$, $n =$ the total number of operand
  - $b$ is output, referred to by $%0$
  - $a$ is input, referred to by $%1$

- “$r$” store in registers
- “$=$” write only
Stack

- **Stack**: work space (memory) for function calls
  - Store arguments, return address, temp variables
  - Function calls: last in, first out (LIFO)
  - Typical usage
    - Caller pushes arguments
    - Caller pushes return address
    - Invokes callee
    - Callee does work
    - Callee pop return address and return
    - ...

x86 instructions to access stack

- X86 dictates that stack grows down

- `pushl %eax` = `subl $4, %esp`  
  `movl %eax, (%esp)`

- `popl %eax` = `movl (%esp), %eax`  
  `addl $4, %esp`

- `call 0x12345` = `pushl %eip`  
  `movl $0x12345, %eip`

- `ret` = `popl %eip`
gcc caller-callee contract on x86

- At entry of callee (i.e., just after call)
  - `%eip` points at first instruction of callee
  - `%esp+4` points at first argument
  - `%esp` points at return address

- After ret instruction
  - `%eip` contains return address
  - `%esp` points at argument pushed by caller
  - `%eax` holds return value
    - `%eax + %edx` for 64 bit, `%eax` trash for void return
  - Called function may have trashed arguments
  - Caller save: `%eax`, `%edx`, and `%ecx` may be trashed
  - Callee save: `%ebp`, `%ebx`, `%esi`, `%edi` must contain contents from time of call
gcc calling convention

- Each function has a stack frame marked by %ebp, %esp
  - %esp can move to make stack frame bigger, smaller
  - %ebp points at saved %ebp from caller, chain

- Function prologue
  - pushl %ebp
  - movl %esp, %ebp

- Function epilog
  - movl %ebp, %esp
  - popl %ebp
gcc calling convention (cont.)

- **Prologue can be replaced by**
  ```c
  enter $0, $0
  ```
  - Not usually used: 4 bytes v.s. 3 for `pusl+movl`, not on hardware fast-patch anymore

- **Epilog can be replaced by**
  ```c
  leave
  ```
  - Usually used: 1 byte v.s. 3 for `movl+popl`
gcc calling convention example

- C code
  
  ```c
  int main(void) { return f(8) + 1; }
  int f(int x) { return g(x); }
  int g(int x) { return x+3; }
  ```
Next lecture

- System call and interrupt