Week 4

* signed 16 bit float memory layout

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | sign bit | exp | | |  |  | | --- | --- | | hidden bit | mantissa | | |  | | --- | | mantissa | | |  | | --- | | mantissa cont. | |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | 0 | 100 | | |  |  | | --- | --- | | 1 | 011 | | |  | | --- | | 0000 | | |  | | --- | | 0000 | |

**^^^ hidden bit is not included in mantissa!!!**

* Assembly
* The control controls the beast
* The memory interface talks between memory and registers
* Calculating units
  + ALU: Arithmetic Logical Unit: bitwise operations and int math
  + FPU: Floating Point Unit: float math
* Registers
  + Def: Bits of memory that hold values temporarily
  + IR: Instruction Register: Holds current instruction
  + IP: Instruction Pointer: Holds address of next instruction
  + SP: Stack Pointer: A pointer to where the local vars are held
  + Flags: Singular bit registers that hold low level, basic info
  + R0-Rn: General Purpose Registers for all sorts of temp vars
* CPUs
  + Big or little endian
  + How many registers? 16, 32, 64?
  + RISC or CISC architecture?
  + Type of machine
    - Stack based : Simple
      * 0 registers specified in instruction
      * Work registers are implicit
      * Calculate (2 \* 3) + (4 \* 5)
        + push 2
        + push 3
        + mult (does 2 pops, mult, push)
        + push 4
        + push 5
        + mult (does 2 pops, mult, push)
        + add (does 2 pops, add, push)
      * Lots of memory used
      * Used in some calculators
    - Accumulator based :
      * 1 register or location specified in instruction
      * Accumulator register is implicit
      * Calculate (2 \* 3) + (4 \* 5)
        + load 2 (into acc)
        + load 3 into r1
        + mult r1 (with acc, result in acc)
        + move into r2 (copied from acc)
        + load 4 (into acc)
        + load 5 into r1
        + mult r1 (mult acc, result in acc)
        + add r2 (with acc, result in acc)
      * Takes less time to access memory
    - General purpose :
    - Up to 2 or 3 registers per instruction
    - No implicit registers other than SP and IP
    - Calculate (2 \* 3) + (4 \* 5)
    - load 2 into r1
    - load 3 into r2
    - mult r1 with r2, result in r2
    - load 4 into r3
    - load 5 into r4
    - mult r3 with r4, result in r4
    - add r2 with r4, result in r4
    - less time to move data between registers
    - Used in most modern computers
  + Intel 4004 1971 4 bit
  + Intel 8080 1974 8 bit
  + Intel 8086 1978 16 bit
    - Had a 16 bit bus to talk to the outside world
    - but so many devices had an 8 bit bus so..
  + Intel 8088 1979
    - Same processor with an 8 bit bus
    - Basis for the original IBM PC 1981
    - Registers
      * AX : General
      * BX : General
      * CX : meant for loops
      * DX : long-term storage
    - 32 bit happens
      * EAX : Double the space, AX takes up half
      * EBX : Double the space, BX takes up half
    - 64 bit happens
      * RAX : Double the space, EAX takes up half
      * RBX : Double the space, EBX takes up half
      * RIP : Lol
* C
  + objdump -d - j .**[filename]** gets a stack dump
* Instruction Pointer holds the addresses of the instructions
* Instructions
  + mov : Take one value from one place and copy it to another place
    - Some CPUs have:
      * load : Load from outside to cpu
      * store : Store from cpu to memory
  + push :
  + pop
  + add / addl
  + imul
  + cmp : Compare but throw away the result
  + and, or, not, xor : int bitwise operations
  + jump, je, goto, etc… : Jumps to an address
* Registers and what they can store
  + **$** denotes a **constant**
  + **%** denotes an **address**
  + Instructions may contain constants: – movl $0x1,%eax “eax = 1;”
  + Instructions often contain registers: – mov %ebx,%eax “eax = ebx”
  + Instructions may refer to memory:
    - mov (%rbp),%eax “eax = mem[rbp]”
      * + “or, eax = \*rbp”
    - mov 0x4(%rbp),%eax “eax = mem[rbp+4]”
    - mov -0xC(%rbp),%eax “eax = mem[rbp-12]”