Contents

1 Summary

1.1 Lectures

- registers
- program counter
- condition codes
- status codes
- processing cycle
- pipelining
- forwarding
- cutting in line
- out-of-order execution

1.2 Labs

1.2.1 Lab 01

- `gcc <file> -o <ouput> -S`: produces an assembly files
- `gcc <file> -o <ouput> -g`: produces a file that is useful for gdb
- `gdb ./<exe> -tui`: start gdb with a graphical view of the code

1.2.2 Lab 02

- `lea <var>(%rip) %<reg>`: load effective address, `<var>(%rip)` 64 bit address of the next instruction, basically loads the memory address of the variable into the specified register
- `xor %eax, %eax`: this sets the return value of a function to 0, `%eax` holds the return values of functions, needs to be set before the function returns

1.2.3 Lab 03

- `mov <var>(%rip) %<reg>`: reads the specified variable from memory and puts it into the register

1.2.4 Lab 04

- `push %rbp`: push the frame pointer of the previous stack frame unto the stack
- `mov %rsp, %rbp`: move the current stack frame address to the frame pointer, %rsp always points to the stop of the stack
- `leave`: undoes the two previous steps
- `mov %rbp, %rsp`, `pop %rbp`: does the same thing as leave

1.2.5 Lab 05

- `sub $0x8, %rsp`: this reserves some space on the stack for local variables to call the functions
- `add $0x8, %rsp`: frees the space that was previously allocated
- `call scanf@plt`: procedural linkage table, contains the address of where scanf is relative to the program, makes function reuse easier

1.2.6 Lab 06

- `call <func>`, `ret`: call pushes the return address onto the stack, return pops it off again to return to where the function was entered
- `cltq`: convert long to quad, basically a cast from `int` to `long` in `C`

1.2.7 Lab 07

- `jmp`: jumps to the label specified, can be used with conditions
- `cmp`: compares two registers, tells if equal, smaller or larger, can be used to condition jump instructions
- different from `call`, this does not push or pop addresses, it just jumps to different parts of the code
- `test` vs `cmp`: `test` is a bitwise `and` while `cmp` is an arithmetic operation
  `test <reg>, <reg> == cmp <reg>, 0`

1.3 TODO

- `.global` labels
- section of assembly code (`.section`)
- order of registers for arguments to functions
- multi-register operations
- `int x 0, 0`

- position independent code
- got (global offset table)
- syscall vs call
- call functions
- jumps
- loops using labels