1 Key Concepts from Labs and Projects

1.1 Lab01 - Hello World in the RISC-V Dev Env

- Console based editing: micro or vim
- git and GitHub usage
- Makefiles
- Hello World in C
- The Autograder

1.2 Project01 - RISC-V VM Access and Number Conversion in C

- Dev Environment Setup
  - Local RISC-V VM using qemu-system-riscv64
  - Remote RISC-V VM on euryale
  - ssh keys, ssh `config`, ssh `authorized_keys`
  - Shell configuration
    - bash: `~/.profile`, `~/.bash_profile`, `~/.bashrc`, `~/.bash_aliases`
    - zsh: `~/.zprofile`, `~/.zshrc`
- C Basics
  - Functions
  - Data (global, stack, heap)
  - Statements and expressions
  - Data types
    - Primitives - `int`,char,float,uint8_t,uint32_t,int32_t,uint64_t,int64_t‘
    - Composit - arrays and structs
    - Pointers - `int *`, `char *`, `uint32_t *`, `uint64_t *`
  - Pointer operations
    - & address of, e.g., `int x; int *p; p = &x;`
    - · dereference, e.g., `x = *p;`
  - Strings in C
    - Array of chars (bytes)
    - Null `\0` (0) terminated
    - ASCII character codes
  - The C stack
    - Local variables
    - Allocated on function entry, deallocated on function exit (return)
  - Signed (`int`, `int32_t`) vs unsigned (`unsigned int`, `uint32_t`) values
- Command line arguments with `int argc` and `char *argv`
  - Layout of `argv` array in memory
  - Array of pointers first, `argv[0]`, `argv[1]`, `argv[2]`, `NULL (0)`
  - String arrays second, `./foo` + `\0`, `-p` + `\0`, etc.
- Parsing command line arguments
- Number systems
  - Decimal (base 10, digits: `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`)
  - Binary (base 2, digits: `0`, `1`, C literal prefix: `0b`)
  - Hexadecimal, "hex" (base 16: digits: `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, `A`, `B`, `C`, `D`, `E`, `F`, C literal prefix: `0x`)
    - `A` = 10, `B` = 11, `C` = 12, `D` = 13, `E` = 14, `F` = 15
- base conversions
  - C string as decimal, binay, hexadecimal to int
    - Place value: `2 * (10^2) + 5 * (10^1) + 4 * (10^0)`
  - int to C string as decimal, binary, hexadecimal
    - Modulus to find digit
    - Divide to get value for next digit

        * Repeat until divide gives 0

1.3 Lab02 - Introduction to RISC-V Assembly Programming

- Instructions, registers, labels, directives
  - See RISC-V Assembly Guide
- Assembling vs compiling
  - Assembling (as) assembly source (.s) to object code (.o)
  - Compiling (gcc) c source (.c) to object code (.o)
  - The object files must be linked, we can use gcc for this
    * `as -o foo.o foo.s`
    * `gcc -o main main.c foo.o`
  - The C compiler can also assemble
    * `gcc -o main main.c main.s`
- Instructions have a name (mnemonic), like `add` or `mul`
- Assembly programming model
  - A RISC-V processor has 32 registers plus the `PC` registers
    * `x0`, `x1`, `... x31`
    * ABI names: `sp`, `ra`, `a0`, `a1`, ..., `t0`, `t1`, ..., `s0`, `s1`, ...
  - The `PC` points to next instruction in memory to execute
  - Load an instruction from memory into processor
  - Execute the instruction, usually updates one or more register values, but can also update memory
  - Update `PC`, often just the next instruction `PC + 4`, but can also be a diffenent address in the case of a branch or jump (control instruction)
- Most instructions have three operands
  - One target (destination)
  - Two source
  - `add t0, t1, t1` means `t0 = t1 + t2`
- On RISC-V 64 bit registers are 64 bits (8 bytes)
- Data processing instructions
- Immediate values
- Control instructions
  - conditional branchs: `beq`, `bne`, `blt`, `bge`
  - jumps: `j`
  - return: `ret`
- Assembly arguments passed in `a0`, `a1`, `a2`, `a3`, etc.
- Return value is put into `a0`
- Basic assembly function

```
.global func_name
func_name:
    add a0, a0, a1
    ret
```

- if/then/else in assembly
  - C:

```
int x, y, r;
r = 0;
if (x > y) {
   r = r + 1;
} else {
   r = r + 99;
}
```

  - Asm: Assume `a0 - int x, a1 - int y, t0 - int r`

```
    li t0, 0
    bgt a0, a1, else
    addi t0, t0, 1
    j endif
else:
    addi t0, t0, 99
endif:
```

- for loops in assembly
  - C:

```
int r, i, n;
r = 0;
n = 10
for (i = 0; i < n; i++) {
    r = r + i;
}
```

  - Assume `t0 - int r, t1 - int i, t2 - int n`

```
    li r, 0
    li n, 10
    li i, 0
loop:
    bge t1, t2, loopend
    addi t0, t0, t1
    addi t1, t1, 1
    j loop
loopend:
```

- Array access in assembly
  - lw (load word) `lw t0, (a0)` means `t0 = *a0`
  - Load word at address `a0` into register `t0`
  - Pointer based array access
    * To get to next element in array of words (ints): `addi a0, a0, 4`

1.4 Project02 - RISC-V Assembly Language

- RISC-V ABI Function Calling Conventions
  - Only one set of registers (32), so we need a way to coordinate usage between functions
  - Caller-saved registers (`a0`, `a1`, ..., `a7`, `t0`, `t`, ... `t6`)
    * These must be preserved on the stack by the caller of a function
    * You only need to preserve the registers you will be using after the function call
  - Callee-saved registers (`sp`, `ra`, `s0`, `s1`, ... `s11`)
    * These must be preserved on the stack by the callee, on entry to the function.
    * They should be restored on exit.

        * The stack pointer is preserved by subtracting (stack allocation) on function entry and adding (stack deallocation) on function exit the name number of bytes.
        * The `sp` should be a multiple of 16 (for performance compatibility with some instructions)

- Functions that don't call other functions do not need to allocate stack space.
    - Only need stack space if the function uses caller-saved registers
- Function call template when using stack:

```
.global func_name
func_name:
    addi sp, sp, -N    # Allocate N bytes on stack, N must be a multiple of 16
    sd ra, (sp)         # Save ra onto stack
                        # Save any callee-saved registers if needed
    ...
    ld ra, (sp)
    addi sp, sp, +N
    ret
```

- Indexed-based array access
    - Assume `t0` is `int i`, `a0` is base address of array `int arr[]`

```
li t1, 4
mul t1, t1, t0
add t1, a0, t1
lw t2, (t1)
```

    - Altneratively use a shift instead of multiply

```
slli t1, t0, 2
add t1, a0, t1
lw t2, (t1)
```

- Recursive functions
    - Nothing special, just that the func calls itself
    - Same rules apply
    - Presere any caller-saved registers on stack before the recursive call
    - Restore any caller-saved registers from stack after the recursive call
    - If a recursive function simply returns after the recursive call, then it is called *tail recursive* and the recursive call can be turned into a jump.