# RISC-V Sim & Single Cycle

Introduction to Computer Systems, Fall 2024

**Instructors:** Joel Ramirez Travis McGaha

**Head TAs:** Adam Gorka Daniel Gearhardt Ash Fujiyama Emily Shen

**TAs:**

| | | |
|---|---|---|
| Ahmed Abdellah | Ethan Weisberg | Maya Huizar |
| Angie Cao | Garrett O'Malley Kirsch | Meghana Vasireddy |
| August Fu | Hassan Rizwan | Perrie Quek |
| Caroline Begg | Iain Li | Sidharth Roy |
| Cathy Cao | Jerry Wang | Sydnie-Shea Cohen |
| Claire Lu | Juan Lopez | Vivi Li |
| Eric Sungwon Lee | Keith Mathe | Yousef AlRabiah |

## Poll Everywhere

pollev.com/tqm

- Any Questions on Registration?

## Logistics

- Midterm: Soon$^{tm}$
  - Edge cases need to be looked at, and a few files need to be rescanned
  - I hope tonight
- HW07 is out and due this Friday at midnight
  - Should have everything you need after this lecture
- HW08 comes out this Friday/weekend
- Check-in06 out later this week
- Looking over the mid sem feedback, will get back to you

## Logistics Pt. 2

- Office Hours this week
  - No Office Hours on Halloween (Thursday)
  - Reduced TA presence at office hours the day after (Friday)

- Will still have lecture on Thursday

- Registration happening soon
  - Plan is to take questions from y'all and then bring some TA's and answers to the last 15 minutes of next lecture

- Election happening soon

## Lecture Outline

- **RISC-V**
- Penn-sim
- Peek at next time
- Aside: Sized Integers
- Binary File I/O

# RISC-V Directives

- We can include directives to indicate where things in our ASM program should be loaded into memory
  - **`.text`**
    - Next instructions are in the .text (code) section of memory
  - **`.data`**
    - Next values are in the .data section space
  - **`.p2align <n>`**
    - Align the next values to the specified power of 2
      .p2align 2 aligns to 4 bytes
  - **`.global <name>`**
    - Registers the name for the assembler and "processor" to recognize outside that file.

- Other directives exist, more on those later

## LC4 Example .asm file

Directives to indicate this is code

```
        .text
        .globl  strlen
        .p2align        2
strlen:
        # comment
        addi x5, x0, 0   # set len (x5) to 0
        lb   x6, 0(x10) # load one bytes from *str into temp
.CONDITION:
        beq  x6, x0, .AFTER  # while (temp != '\0') {
        addi x5, x5, 1       #   len += 1
        addi x10, x10, 1     #   str += 1
        lb   x6, 0(x10)      #   temp = *str
        jal  x0, .CONDITION  # }
.AFTER:
        addi x10, x5, 0      # puts len into the "return value" reg
.END:

        .data
        .p2align        0
        .global str
str:
        .asciz "hello"   # ascii characters with a '\0' at the end
```

Goes back to .CONDITION

## Poll Everywhere

pollev.com/tqm

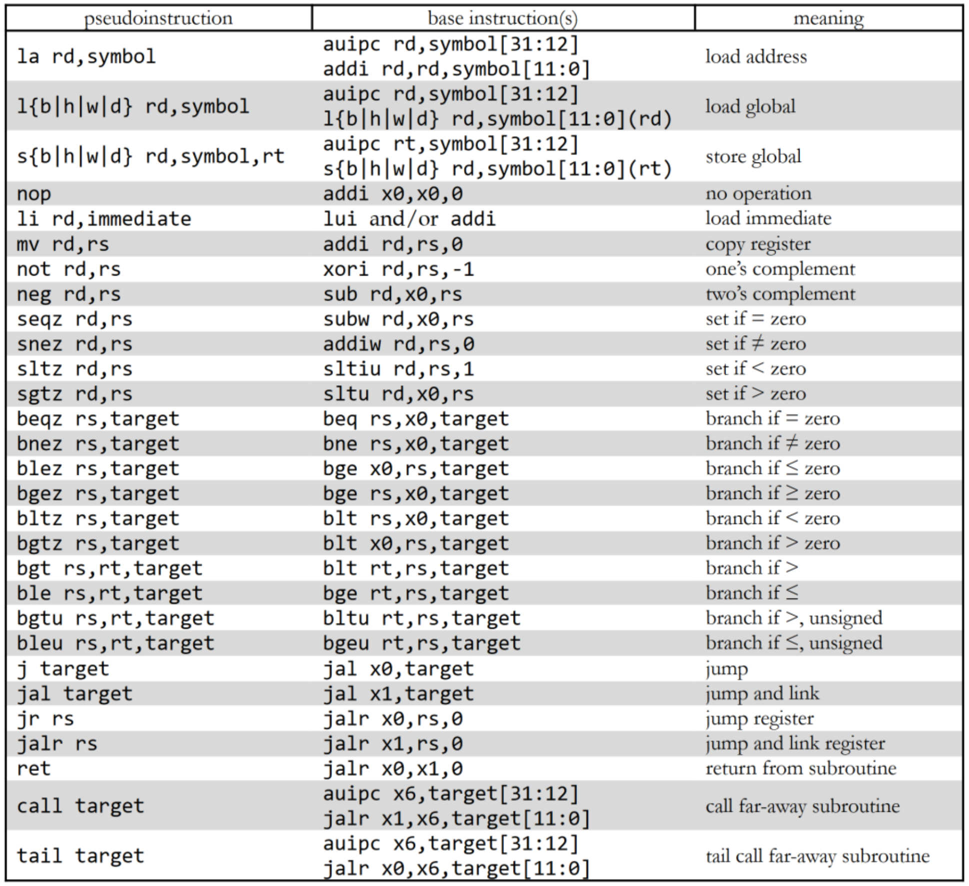
- We have a lot of branch instructions, but we don't have a **`ble`** (Branch if less than or equal to [<=]) instruction.

  Why is a **`ble`** instruction not needed?

| INSTR | Meaning |
|---|---|
| beq rs1, rs2, imm12 | Branch if rs1 == rs2 |
| bne rs1, rs2, imm12 | Branch if rs1 != rs2 |
| blt rs1, rs2, imm12 | Branch if rs1 < rs2 |
| bge rs1, rs2, imm12 | Branch if rs1 >= rs2 |
| bltu rs1, rs2, imm12 | Branch if rs1 < rs2 (treat values as unsigned) |
| bgeu rs1, rs2, imm12 | Branch if rs1 >= rs2 (treat values as unsigned) |

## Pseudo-Instructions & Register Aliases

- There are a lot of common operations that don’t translate directly into instructions, but we can do the equivalent thing in one “real” instruction.
- Pseudo instructions will be translated into their base instruction(s) by the "compiler".
- The processor has no notion of these pseudo-instructions.



| pseudoinstruction | base instruction(s) | meaning |
|---|---|---|
| `la rd,symbol` | `auipc rd,symbol[31:12]`<br>`addi rd,rd,symbol[11:0]` | load address |
| `l{b\|h\|w\|d} rd,symbol` | `auipc rd,symbol[31:12]`<br>`l{b\|h\|w\|d} rd,symbol[11:0](rd)` | load global |
| `s{b\|h\|w\|d} rd,symbol,rt` | `auipc rt,symbol[31:12]`<br>`s{b\|h\|w\|d} rd,symbol[11:0](rt)` | store global |
| `nop` | `addi x0,x0,0` | no operation |
| `li rd,immediate` | `lui and/or addi` | load immediate |
| `mv rd,rs` | `addi rd,rs,0` | copy register |
| `not rd,rs` | `xori rd,rs,-1` | one’s complement |
| `neg rd,rs` | `sub rd,x0,rs` | two’s complement |
| `seqz rd,rs` | `subw rd,x0,rs` | set if = zero |
| `snez rd,rs` | `addiw rd,rs,0` | set if ≠ zero |
| `sltz rd,rs` | `sltiu rd,rs,1` | set if < zero |
| `sgtz rd,rs` | `sltu rd,x0,rs` | set if > zero |
| `beqz rs,target` | `beq rs,x0,target` | branch if = zero |
| `bnez rs,target` | `bne rs,x0,target` | branch if ≠ zero |
| `blez rs,target` | `bge x0,rs,target` | branch if ≤ zero |
| `bgez rs,target` | `bge rs,x0,target` | branch if ≥ zero |
| `bltz rs,target` | `blt rs,x0,target` | branch if < zero |
| `bgtz rs,target` | `blt x0,rs,target` | branch if > zero |
| `bgt rs,rt,target` | `blt rt,rs,target` | branch if > |
| `ble rs,rt,target` | `bge rt,rs,target` | branch if ≤ |
| `bgtu rs,rt,target` | `bltu rt,rs,target` | branch if >, unsigned |
| `bleu rs,rt,target` | `bgeu rt,rs,target` | branch if ≤, unsigned |
| `j target` | `jal x0,target` | jump |
| `jal target` | `jal x1,target` | jump and link |
| `jr rs` | `jalr x0,rs,0` | jump register |
| `jalr rs` | `jalr x1,rs,0` | jump and link register |
| `ret` | `jalr x0,x1,0` | return from subroutine |
| `call target` | `auipc x6,target[31:12]`<br>`jalr x1,x6,target[11:0]` | call far-away subroutine |
| `tail target` | `auipc x6,target[31:12]`<br>`jalr x0,x6,target[11:0]` | tail call far-away subroutine |

# Pseudo-Instructions & Register Aliases

- We have 32 registers (x0 through x31) and the program counter.
- Some registers have conventional uses.
  - x5 is usually used as a “temporary” register (t0)
  - x10 is usually the first argument to a function (a0)
    - Sometimes is also where return value is stored
  - x11 is usually the first argument to a function (a1)
  - etc.
  - **PC always is the PC**

## More Sample RISC-V Program

- Let’s look at some more risc-v programs
  - `strlen.s` (the program I showed earlier)
  - `sum_numbers.s`
  - `multiply.s`

# More RISC-V Directives

- These assembly directives provide information on how various data/constants should be assembled
  - **`.word <value>`**
    - Store the associated 32-bit value in memory
  - **`.half <value>`**
    - Store the associated 16-bit value in memory
  - **`.fill <repeat>, <size>, <value>`**
    - Stores <repeat> values of <size> size contiguously in memory
  - **`.asciz "string value"`**
    - Store the associated string value (with a null terminator) in memory
  - **`.ascii "string value"`**
    - Same as before, but with no null-terminator

## Lecture Outline

- RISC-V
- **Penn-sim**
- Peek at next time
- Aside: Sized Integers
- Binary File I/O

## ASM Files

- Assembly Files are text files that contain a lot of conveniences for ASM programmers
  - Instructions written as text (not as 32-bit patterns)
  - Comments
  - Initial values of some memory locations
  - Labels to refer to addresses and values
  - Directives
  - Pseudo Instructions

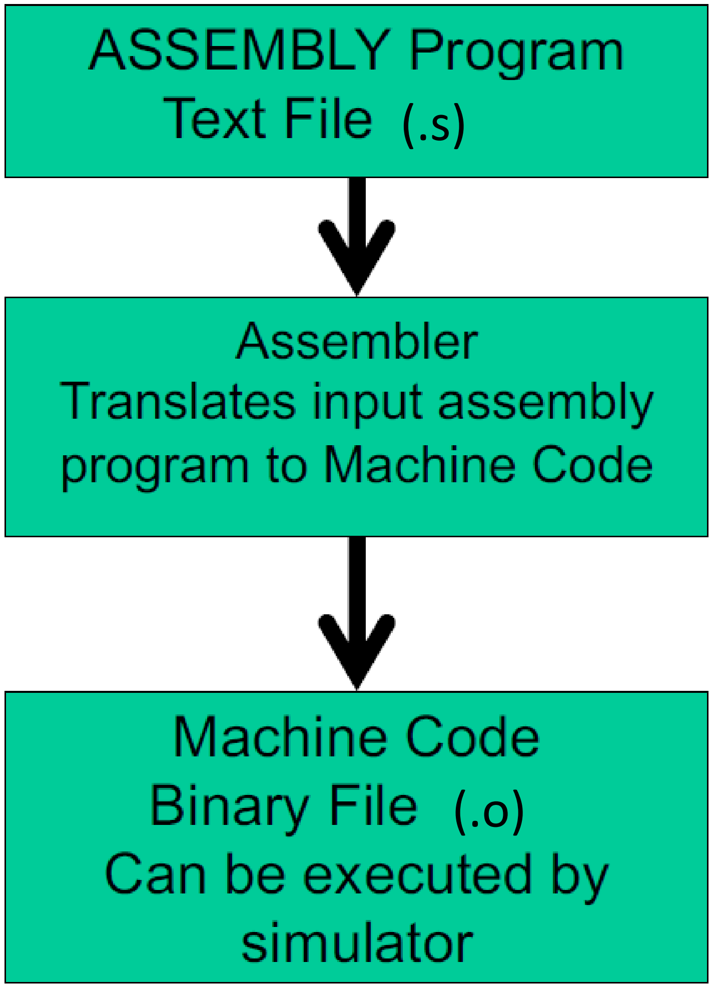
- ASM files are not directly run on a processor, this file needs to be processed into something machine-readable

## Text Files

- `.s` (sometimes `.asm`) files and source code files for most languages are text files:
  - They can only contain ascii (or sometimes Unicode) characters.
  - Are not directly executed by the computer
- Text files are different from .docx and .pdf files
  - You cannot write text files in Microsoft Word
- Text files are created and edited by a text editor
  - Vim, Emacs, Notepad, Notepad++, Nano, Sublime, Atom, etc.

## ASM -> OBJ Process

- Assembly Files need to be processed by an assembler to become machine code
- Machine code can be executed directly by the computer hardware.
  (or a simulator in our case)
- Demo: `multiply.o` vs `multiply.s`


ASSEMBLY Program
Text File (.s)
Assembler
Translates input assembly program to Machine Code
Machine Code
Binary File (.o)
Can be executed by simulator

## penn-sim

- single_as: A C++ program written to:
  - Convert a single RISC-V Assembly file to an object file w/ machine code
- penn-sim: A C++ program written to:
  - Simulate the operations of RISC-V ISA
  - Provide debugging tools for RISC-V ISA
- penn-sim Demo
  - (See the lecture recording)
- Travis wrote these this semester. If there is an error, let him know.

## penn-sim Commands Pt. 1

- penn-sim has a command line at the top that you can type commands into
  - `set <register> <value>`
    - Loads a specific value into a register, works for special registers too (PC)
  - `list`
    - Lists the ASM instructions around the current program couter
  - `info registers`
    - Lists out the current values of all registers
  - `info labels`
    - Lists out the current values of all labels
  - `help`
    - Gives some helpful information on the what commands are available and what they do.

## PennSim Commands Pt. 2

- `break <label | address>`
  - Set a break point at the specific label or address
- `del <label | address>`
  - Remove a break point at the specific label or address
- `step`
  - Simulate the execution of the next instruction
- `run`
  - Continue the simulation until a breakpoint or fatal error is encountered

- `script <filename>`
  - You can put a sequence of commands in a text file and then run them all at once using this command. Convenient for HW07

## Lecture Outline

- RISC-V
- Penn-sim
- **Peek at next time**
- Aside: Sized Integers
- Binary File I/O

## Poll Everywhere

pollev.com/tqm

- What is the purpose of this assembly?

```
        .text
        .globl   foo
        .p2align        2
foo:
        mv       a4,a0
        li       a5,1
        mv       a0,a5
        ble      a4,a5,.L7
.L6:
        mul      a0,a0,a5
        addi     a5,a5,1
        bne      a4,a5,.L6
.L7:
        ret
```

## Poll Everywh

- Wanna guess this one?

```
mystery:
        blt     a0,zero,.L10
        addi    sp,sp,-16
        sw      ra,12(sp)
        sw      s0,8(sp)
        mv      s0,a0
        addi    a5,a0,-1
        li      a4,1
        mv      a0,a4
        bleu    a5,a4,.L8
        sw      s1,4(sp)
        mv      a0,a5
        call    mystery
        mv      s1,a0
        addi    a0,s0,-2
        call    mystery
        add     a0,s1,a0
        lw      s1,4(sp)
.L8:
        lw      ra,12(sp)
        lw      s0,8(sp)
        addi    sp,sp,16
        jr      ra
.L10:
        li      a0,0
        ret
```

## Next Lecture: Hardware/Software Interface

- We've looked at some hardware topics and some software topics (C & RISC-V)

- Thursday we are looking at the hardware/software interface for the RISC-V ISA
  - How does assembly run on hardware?
  - How do we create hardware that runs assembly code?

- Hardware details abstracted, uses a lot of the components previously talked about (Mux, Adders, Incrementors, etc.)
  - You will implement something like this in CIS 4710

## HW/SW Interface

- Assembly is a middle ground between software and hardware

- Software:
  - It can still be very hard to read assembly code
  - More complex coding languages translates into assembly

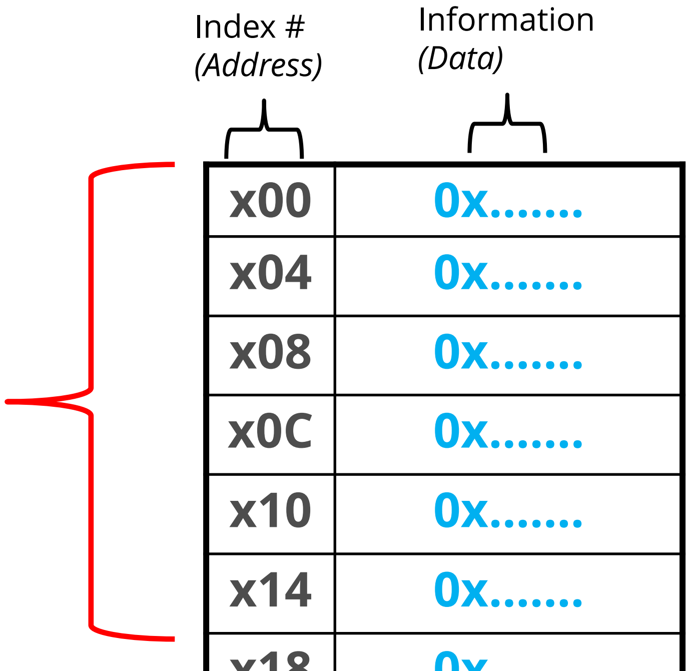
- Hardware:
  - Hardware only needs to implement these “simple” instructions.
  - Hardware does not *need* to implement a custom “calculate the Fibonacci sequence” piece of hardware.
  - Instructions translate directly into binary that hardware can read.

# Reminder: Instructions are bits

- An instruction fits in 32-bits (4-bytes, 4 memory locations)
- These instructions are stored in memory and accessed sequentially
  - When we trace through the code, we are just accessing the next instruction in memory

```
li x0, #32
li x1, #16
li x2, #64

div x1, x2, x1
add x3, x3, x0
sub x0, x2, x3
```

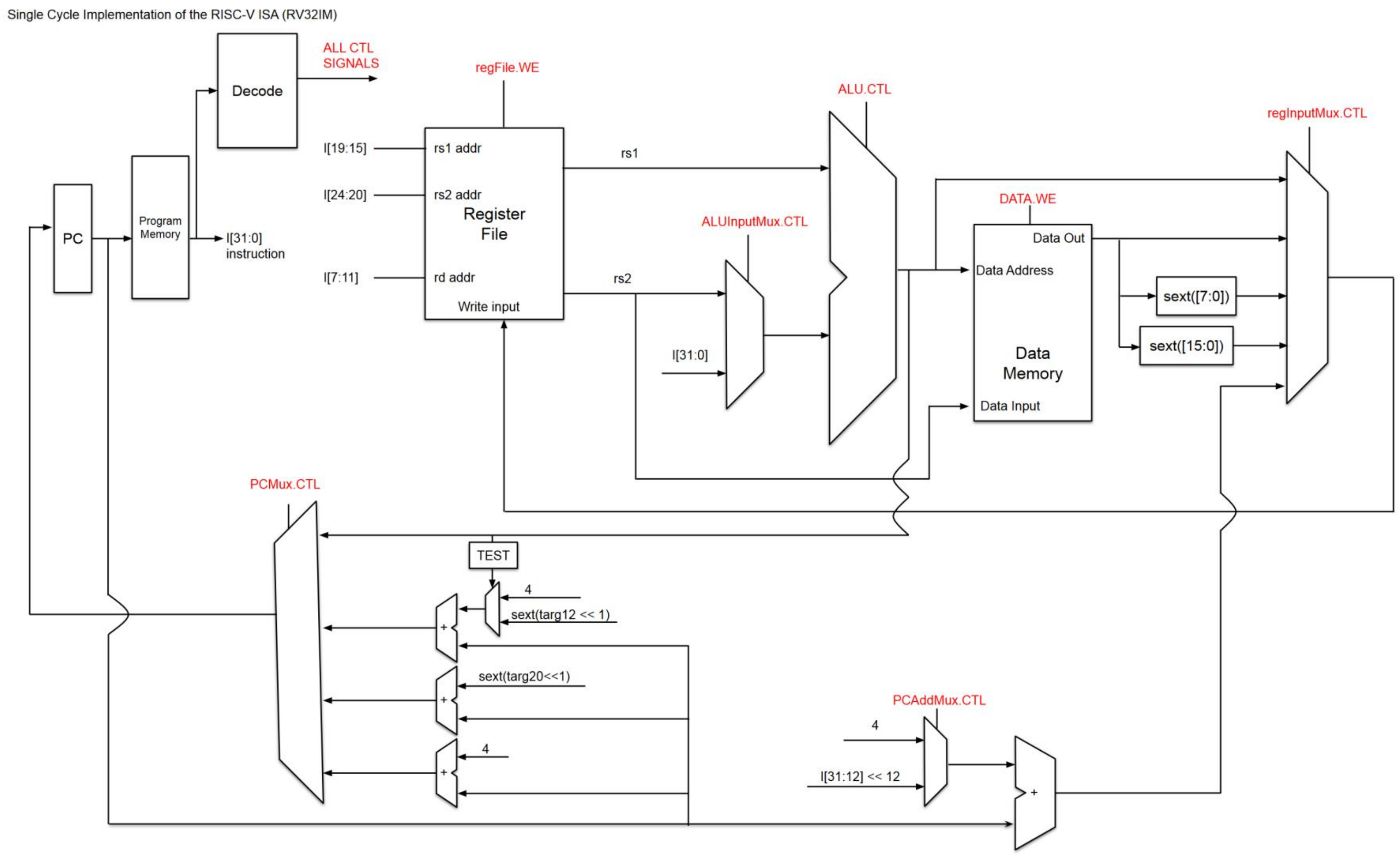


## The Von Neumann Loop

- Von Neumann Processor essentially does:
  - Fetch instruction at Program Counter
  - Decode instruction
  - Execute instruction & Update PC
  - Repeat

- Critical Requirement
  - Each iteration of this loop must appear **atomic** (All or nothing)
  - Key word from programmer perspective: atomic
    - Maintains sanity
  - Key word from hardware perspective: appear
    - Enables hardware to perform various tricks for performance >:]

## An Idea of what we are doing next lecture:



## More RISC-V References!

- More RISC-V References added to the course website
- Highly recommend you print out a copy of the “Control Signals Description” handout

- RISC-V Single Cycle Processor is the diagram on the previous slide, may also want to print this

## Lecture Outline

- RISC-V
- Penn-sim
- Peek at next time
- **Aside: Sized Integers**
- Binary File I/O

## Aside: sized integer types

- Quick question: How big is an `int`?

## Aside: sized integer types

- Quick question: How big is an `int`?
- For what we have said, an int is 4-bytes (32-bits)…
  but this is not something guaranteed by the C standard.

| Type | Minimum Size |
|---|---|
| char | 8-bits |
| short | 16-bits |
| int | 16-bits |
| long | 32-bits |
| long long | 64-bits |

## Aside: sized integer types

- The C standard library provides `#include <stdint.h>`

- Provides types that we can guarantee the size of:
  - `int32_t       //  32-bit signed integer`
  - `uint32_t      // 32-bit unsigned integer`
  - `int16_t`
  - `uint16_t`
  - `uint8_t`
  - etc.

- You will see these on the next homework assignment

## Lecture Outline

- RISC-V
- Penn-sim
- Peek at next time
- Aside: Sized Integers
- **Binary File I/O**

## File I/O

- In the past, we have talked about memory, which is often a type of “volatile storage”
  - Volatile storage: requires power to maintain the data values. Loss of power = loss of data
  - Program memory is also deallocated at the end of the program.
  - To get those values again, the program to compute them must be run again

- Files: a type of permanent/long-term “non-volatile” storage
  - Non-volatile: retains data when the power is turned off
  - Long-term: holds data that exists beyond the lifetime of a program

## File: Examples

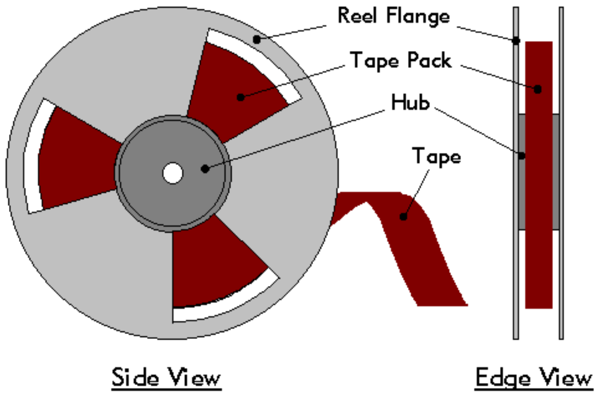
- You’ve already been interacting with files

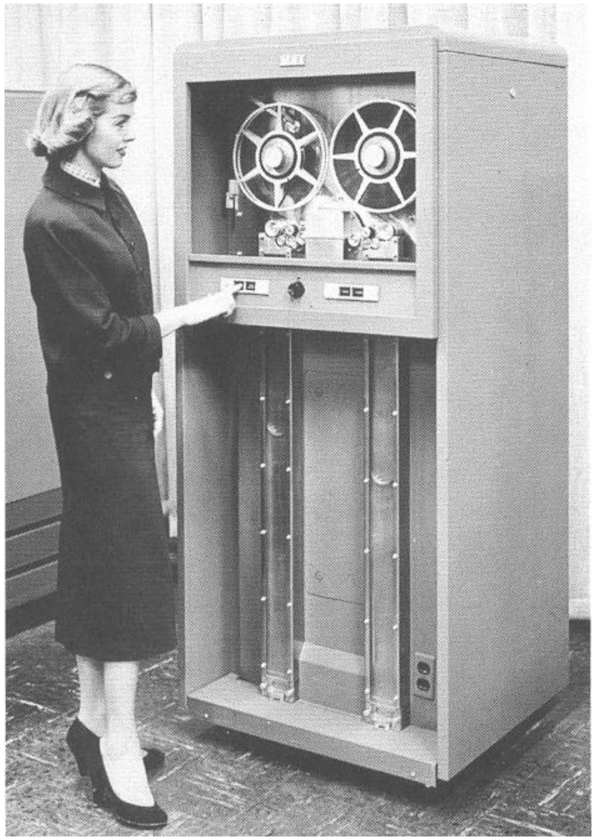
- Program files: (.c/.asm/.obj/etc.) are modified and persist between program executions.
  - While these contain information about how a program is setup, it doesn’t contain all of program memory, which will change as the program executes

- Editors (sublime/IntelliJ/vim/PowerPoint/etc.) are programs that read and modify files based on user input

## File Interface Metaphor: Tape Drives

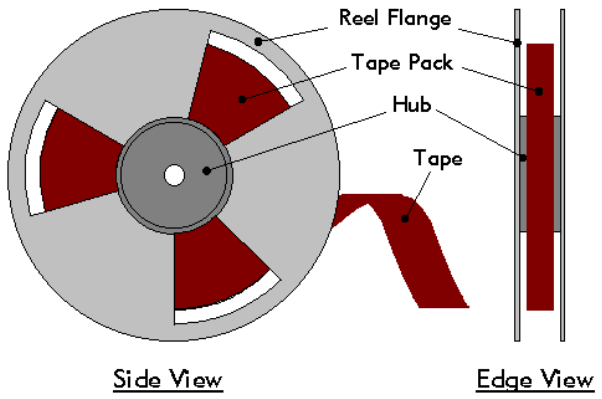
- Programs usually interact with files following a similar file interface:
  - Functions that model a sequential access device like magnetic tape drives

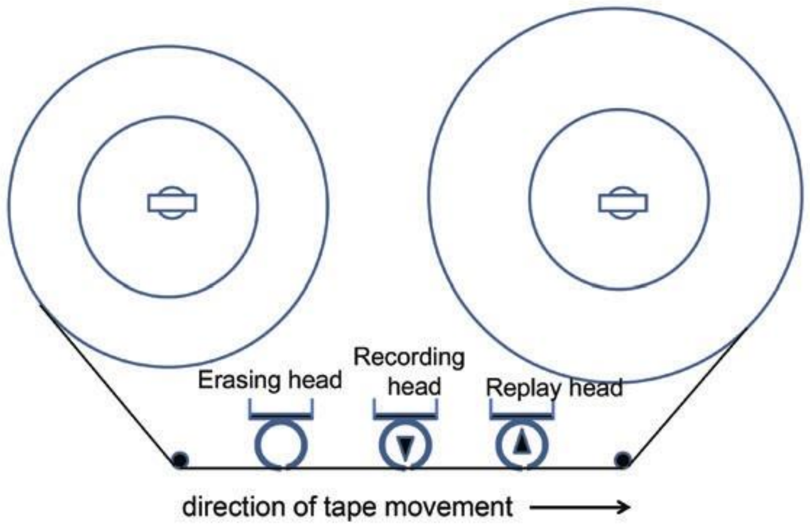




# File Interface Metaphor: Tape Drives

- Example File Operations:
  - Open a file for reading or writing (usually starting at the beginning of the file)
  - Read/Write the file
    - Each read/write advances the number of bytes read or written
  - Rewind: start at beginning again
  - Others

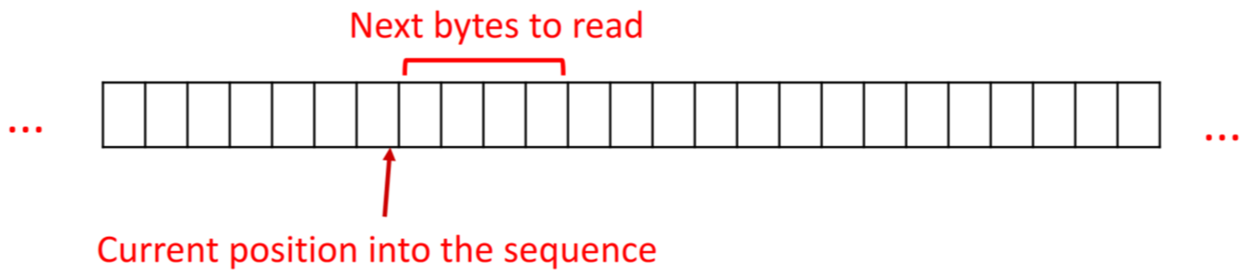




## File Interface Metaphor: Streams

- Another (more modern) abstraction is to think of I/O in terms of "streams"

- Stream:
  - A sequence of bytes that flows to and from a device
  - We do not have access to whole file at once
    - (some files are too big to fit inside of memory easily)
    - Files are not stored entirely contiguously.



Another way to think of this is that it is an "iterator" over the file contents (sort of).
The iterator can read the current data, overwrite the current data, reposition, etc.

## C Stream Functions (1 of 3)

- Some stream functions (complete list in `stdio.h`):

  Returns NULL on error

  Do we create a new file if it doesn't exist?

  Are we reading the file?

  Are we writing the file?

  - `FILE* fopen(filename, mode);`
    - Opens a stream to the specified file in specified file access mode
  - `int fclose(stream);`

    a **FILE*** returned by fopen
    - Closes the specified stream (and file)
  - `int fprintf(stream, format, ...);`
    - Writes a formatted C string
      - Like `printf(...);` but for files
  - `int fscanf(stream, format, ...);`
    - Reads data and stores data matching the format string

# C Stream Functions (2 of 3)

- Some stream functions (complete list in `stdio.h`):

Pointer to the start of elements in memory to write to file

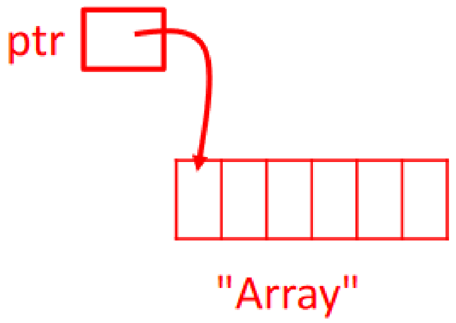
Size of an element

Number of elements

**FILE***

  - `size_t fwrite(ptr, size, count, stream);`
    - Writes an array of *count* elements of *size* bytes from *ptr* to *stream*
  - `size_t fread(ptr, size, count, stream);`
    - Reads an array of *count* elements of *size* bytes from *stream* to *ptr*

Returns number of elements actually read/written



## C Stream Functions (3 of 3)

- Some stream functions (complete list in stdio.h):

  -
    ```c
    int fgetc(FILE* stream);
    ```
    - Reads one character (one byte)

  -
    ```c
    int fputc(char c, FILE* stream);
    ```
    - Prints one character (one byte)

  -
    ```c
    char* fgets(char* str, int n, FILE* stream);
    ```
    - Reads a string from the strearm into the string **`str`**. Reads N characters or until a newline character (or end of file).

## C Stream Error Checking/Handling

- Some error functions (complete list in `stdio.h`):

  -
    ```
    int ferror(stream);
    ```
    - Checks if the error indicator associated with the specified stream is set

  -
    ```
    int clearerr(stream);
    ```
    - Resets error and EOF indicators for the specified stream

  -
    ```
    void perror(message);
    ```
    - Prints `message` followed by an error message related to `errno` to `stderr`

Global variable

Extra information

## Terminal input/output

- C defines three file streams for terminal input/output
  - Defined in <stdio.h>
  - Opened at program start by default

  - stdin: standard input (console)
  - stdout: standard output (console, for normal output)
  - stderr: standard error (console, for error output)

- The following are equivalent

```
printf("Hello World!\n");
```

```
fprintf(stdout, "Hello World!\n");
```

## Demo: copy file program

- Well Written file posted on website as copy_file.c

- Things to do when dealing with C stream I/O
  - Eventually we will hit the end of file, need to handle that
  - Must ask for a number of bytes/elements to be read.
  - If possible, best practice is to request for a chunk of bytes/elements at a time (Not applicable in this class)

## Other Functions

- Many other functions not covered in lecture (not enough time). Feel free to look up others and use them

- Some examples:
  - int feof(FILE* f);
    - check for end of file
  - void rewind(FILE *f);
    - start back at the beginning of file
  - long ftell(FILE* f);
    - gives the current position into the file
  - int fseek(FILE* f, long offset, int whence);
    - Reposition where we are in the file

## Binary files & Serialization

- So far this lecture has implicitly assumed we are working with files that hold text (characters)

- Binary files also exist where data isn't stored as characters. (object files are an example)

- Some data/data-structures make more sense to be stored in binary through a process called **serialization.**

## Serialization Example:

- Posted on course website
  - read_floats.c
  - write_floats.c

- Notes:
  - Don’t have to read/write an array, can read/write only one “element”
  - Trying to open these files in an editor will not be readable

## Endianness

- There is one byte at each address location
  - For multi-byte data, how do we order it in memory?
  - Data should be kept together, but what order should it be?

  - Example, store the 4-byte (32-bit) int:
    0x A1 B2 C3 D4

Each byte has its own address

Most significant Byte

Least significant Byte

- The order of the bytes in memory is called endianness
  - Big endian vs little endian

Note how the hex digits within a byte are still in the same order

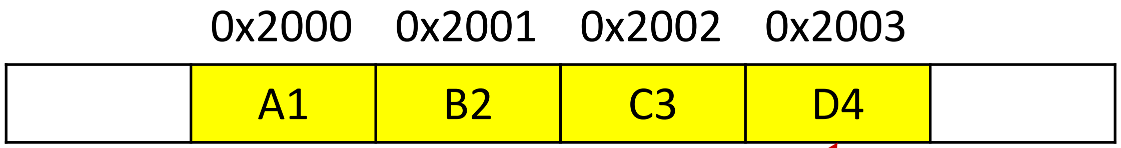
# Endianness

Each byte has its own address

- Consider our example 0x A1 B2 C3 D4

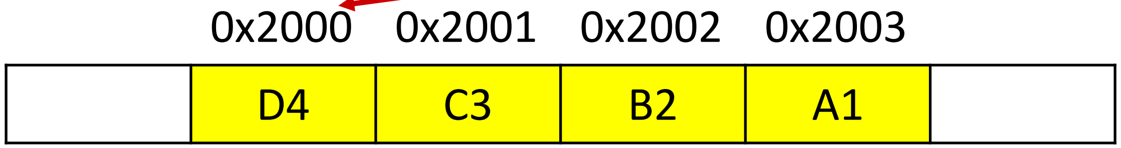
Least significant Byt

- Big endian
  - Least significant byte has highest address
  - Looks the most like what we would read
  - The standard for storing information on files/the network



- Little Endian
  - Least significant byte has lowest address
  - What your VM probably uses

Least significant byte



## Poll Everywhere

pollev.com/tqm

- If we have the following int which is four bytes. on a big-endian machine, how would this be stored in memory?

```
int num = 0xCADEDADA;
```

A.

| | | | |
|---|---|---|---|
| CA | DE | DA | DA |

B.

| | | | |
|---|---|---|---|
| DA | DA | DE | CA |

C.

| | | | |
|---|---|---|---|
| AC | ED | AD | AD |

D.

| | | | |
|---|---|---|---|
| AD | AD | ED | AC |

E. I'm not sure

## Poll Everywhere

pollev.com/tqm

- If we have the following int which is four bytes. on a big-endian machine, how would this be stored in memory?

```
int num = 0xCADEDADA;
```

| | | | | |
|---|---|---|---|---|
| [✓ A.] | CA | DE | DA | DA |
| B. | DA | DA | DE | CA |
| C. | AC | ED | AD | AD |
| D. | AD | AD | ED | AC |

E. I'm not sure

## Endianness: Why it matters

- Since machines may store things in different byte orderings, it causes problems when they share files or communicate over the network.
- A standard ordering is used for storing binary data, big endian (often called Network ordering).
- Need to make sure that we store bytes in network byte ordering when we serialize data

## Endianness functions

- There are some functions out there that convert byte orderings
  - htons() -> Host to Network short (16 bits)
    - Converts from Host byte ordering to network byte ordering
  - ntohs() -> Network to Host short (16 bits)
    - Converts from network byte ordering to host byte ordering
- "Network byte order" is big endian. Your "host" machine is little endian
- More info in <arpa/inet.h>
- ▪ Variants also exist for 32 bit conversion

## Practice Time

- Finish implementing the following C function:

```
uint16_t ntohs(uint16_t to_convert) {
  uint16_t res = 0;

  // you may find these useful:
  // 0xFF (eight 1 bits integer constant)
  //   |  (bitwise or)
  //   &  (bitwise and)
  //   >> (right shift)
  //   << (left shift)

  return res;
}
```

- What would the reverse look like? `htons()`