**CS 241 Midterm Review**

* **MIPS programming**
  + Memory in RAM is organized in words – each word = 4 bytes = 32 bits
    - Addresses for each word are 0x00, 0x04, 0x08, etc.
  + 32 registers on the CPU – $0 to $31
    - Each register stores one word
    - $0 = always 0
    - $30 = stack pointer (SP)
      * Stack is full descending – SP points to the topmost element in stack
      * Stack grows from higher → lower address
        + i.e. decrement SP to push to stack, increment SP to pop from stack
    - $31 = holds address of next instruction (program) in the loader; i.e. return address
    - Program counter (PC) = holds address of next instruction to be executed
    - Instruction register (IR) = holds current instruction being executed
    - Fetch-execute cycle:
      * IR ← MEM[PC] ;get instruction from memory pointed to by PC
      * PC ← PC + 4 ;increment PC to point at next instruction
      * Execute instruction in IR
  + Memory operations:
    - lw/sw $a, i($b) – target address is $b + i
    - Load word
      * Put address to load from in memory access register (MAR)
      * Retrieves data from memory at that address
      * Data is placed in memory data register (MDR)
      * Data is moved from MDR to destination register
    - Store word
      * Put address to store to in MAR
      * Move data from source register into MDR
      * Send data to memory and store value
  + Arithmetic operations
    - add/sub $a, $b, $c
    - mult/div/multu/divu $1, $2 – result stored in hi:lo
      * Use mfhi/mflo to retrieve result from hi/lo
  + lis $d, .word i
    - Load immediate (numeric) value into $d
    - Or load a label – actually stores the address of line that the label is on
    - lis automatically loads and skips next line, i.e. PC += 8 after lis
  + Branching
    - beq/bne $a, $b, i – i is a relative offset
      * i can be a label – actually calculates & stores the offset between PC and label
  + jr $31
    - Finishes/returns program; sets PC = $31 (return address of program)
  + Subroutines
    - In MAIN:
    - First push current $31 value onto stack
    - Pass parameters into/out of subroutine with registers
    - lis $d, .word SUB – load address of label SUB into $d
    - jalr $d – calls subroutine; saves PC (next instruction in main) in $31, sets PC= $d (first instruction of SUB)
    - Pop $31 from stack after returning from subroutine
    - …
    - In SUB:
    - Save registers that will be used in subroutine
      * sw $2, -4($30)
      * sw $3, -8($30) ;push each register onto stack
      * sub $30, $30, $9 ;adjust SP for push ($9 is the # of values saved \* 4)
      * …
      * add $30, $30, $9 ;adjust SP for pop
      * lw $3, -8($30)
      * lw $2, -4($30) ;pop each register from stack
    - jr $31 – sets PC back = $31 (returns to main)
  + Output
    - sw to address 0xffff000c to output the last byte of data as a character on-screen
      * Characters are encoded using ASCII
* **The assembler**
  + Assembly language → assembler → machine code
  + Pass 1 – analysis
    - Tokenize the input
    - Checks for syntax errors – improper form/structure of code
    - Checks for semantic errors – duplicate label definitions
    - Outputs intermediate representation – comments removed & tokenized code input
    - Outputs symbol table – map of (label, address) pairs
  + Pass 2 – synthesis
    - Translates each instruction into machine code
    - Replace labels with corresponding addresses (lookup in symbol table)
    - Output machine code in bytes (8 bits/2 hex digits → 1 char; 1 word → 4 chars)
      * i.e. shift word by 8 bits 4 times
* **Loader**
  + Loads program from secondary storage (hard drive) into primary storage (RAM) and executes it
  + Determines length of program, finds & allocates space in RAM to copy program into
  + Loads program at starting memory address = α
* **MERL format**
  + Header
    - Cookie = 0x10000002 (beq $0, $0, $2)
    - File length = header + code + footer → points to line after last line of file (endFile:)
    - Code length = header + code → points to first line of footer (endCode:)
  + MIPS code
    - Starts at address α + 0x0c
  + Footer
    - Relocation; i.e. .word LABEL
      * Word 1 = 0x01
      * Word 2 = address of relocatable word (address where LABEL is used)
    - External symbol definition (ESD); i.e. label is used in another file
      * Use directive: .export LABEL
      * Word 1 = 0x05
      * Word 2 = address represented by label
      * Word 3 = length of label in chars
      * Word 4+ = each char in label in ASCII (1 char per word)
    - External symbol reference (ESR); i.e. label is defined in another file
      * Use directive: .import LABEL
      * Word 1 = 0x11
      * Word 2 = address where label is used
      * Word 3 = length of label in chars
      * Word 4+ = each char in label in ASCII (1 char per word)
* **Loader relocation algorithm**
  + For each relocate → relocate.address += 12 if code doesn’t already start at 0x0c
  + For each relocate → relocate.address += α (adjust pointer to where label is used)
    - → label at relocate.address += α (adjust pointer to where label is defined)
* **Linker relocation algorithm**
  + (header 1, code 1, footer 1) + (header 2, code 2, footer 2) → (header, code1, code2, footer)
  + α for m2 = m1.code\_length – 12
  + For each m2.relocate → relocate.address += α
  + For each m2.export → export.address += α
  + If m1 & m2 are exporting the same labels → ERROR
  + If a label is imported & exported in the same file → ERROR
  + If any label is not found in file and no .import for it → ERROR
  + For each m1.import == m2.export →
    - Replace import in m1 (should be 0) with export.address
    - Remove import from m1.imports
    - Add import to m1.relocates
  + For each m2.import == m1.export →
    - Replace import in m2 with export.address
    - Remove import from m2.imports
    - Add import to m2.relocates (relocate by + α)
  + Recalculate file length, code length
  + Combine m1 & m2 relocates, imports, & exports
    - ESDs should be preserved; remove all resolved ESRs
* **Formal languages (regular languages)**
  + Provide precise specifications for a language – describe its expressiveness and limitations
  + Alphabet = ∑ = finite set of symbols
  + String/word = finite sequence of symbols from ∑
    - ε = empty word
  + Language = L = set of words
  + Regular language can be:
    - ∅ = empty language
    - {a} = singleton language
    - L1 ∪ L2 = union of 2 languages
    - L1 ⋅ L2 = concatenation of 2 languages
    - Ln = language consisting of n words from L
    - L\* = repetition of a language; i.e. 0 or more occurrences of words from L
      * L\* = ε ∪ L1 ∪ L2 ∪ L3 ∪ …
  + Regular expression vs. set notation
    - (ab) = {a}{b} = concatenation of words (a followed by b)
    - (a|b) = {a, b} = union of words (a or b)
    - a\* = {a}\* = repetition of words (0 or more occurrences of a)
  + Regular expression shorthands
    - [a-d] = (a|b|c|d)
    - a+ = aa\* (i.e. repetition excluding ε)
    - . = any single character
* **Finite automata**
  + Deterministic finite automata (DFA) = (∑, Q, q0, A, δ)
    - ∑ = finite alphabet
    - Q = finite set of states
    - q0 ∈ Q = start state
    - A ⊆ Q = set of accepting states
    - δ : Q × ∑ → Q = transition function δ(state, symbol) → next state
    - Deterministic = each transition leaving a state has a unique symbol
      * A unique path to each accepting state
    - If input ends before an accepting state is reached → ERROR
    - If transition for input doesn’t exist → ERROR (implicit error state)
  + Non-deterministic finite automata (NFA)
    - δ: Q × ∑ → 2Q = power set (all subsets) of Q
    - Non-deterministic = transitions leaving the same state can have the same symbol
      * Multiple paths to each accepting state
  + ε-transition – transition between states that require no input
  + ε-NFA – can have ε-transitions
  + Converting regular expression → ε-NFA
    - Link with ε-transitions
    - Concatenation (E1E2) – link accepting states of E1 → starting state of E2
    - Union (E1|E2) – link new start state → start states of E1 & E2
    - Repetition (E\*) – link accepting states → start state
  + Converting NFA → DFA (subset construction)
    - For every DFA state q = ∪(NFA states) = {p1, p2, …}:
      * δDFA(q, transition) = ∪(δNFA(p, transition))
      * Repeat until every DFA state has a transition for every letter in ∑
    - Make DFA state accepting if it contains any accepting states in the NFA
  + Maximal munch scanning – keep reading input until an error state is reached
    - If last state was non-accepting → ERROR
    - If last state was whitespace → ignore
    - If last state was accepting → output (valid token)
    - Go back to start state and scan next token