Midterm 2 Cheat Sheet

RISC-V Architecture

- Register File: Stores temp vars. Each read port uses read_index(n bits, input) and read_data(m bits, output). Each write port uses write_index(n bits, input), write_data(m bits, input), write_enable(1 bit, input).
- RISC-V Register File: 32x32 (n = 5, m = 32), 2 read ports, 1 write port.
- RISC-V: 32-bit memory model, max 2^32 bytes of memory or 4GB. Memory range: 0x00000000 to 0xFFFFFFFF. Instructions (text), data, stack all in memory. Global/static allocated before runtime, dynamic/heap during runtime.
- RISC-V ISA: Built-in data types, fixed instruction set, memory model. 6 instruction formats (R, I, S, B, U, J).
- Memory and registers: Word is 32 bits or 4 bytes. Registers are 32 bits. Memory is byte addressable.
- Instruction explanation: op (7 bits, operation), funct7 (7 bits, augments op), funct3 (3 bits, augments op), rs1 (5 bits, first register source), rs2 (5 bits, second register source), rd (5 bits, register destination), imm (12 bits, immediate value or constants, usually sign extended).

RISC-V Assembly

- Directives: .data (data section), .text (text section), .globl (global variable), .word (word), .asciiz (string)
- Labels: Mark/jump to program location
- System calls: Invoke OS service
- Registers: Use x0 or zero, ra or x1, etc. x0, zero always 0. Saved registers store vars, temporary registers store temp vars.
- Byte addressing: 32 bit word, addresses divisible by 4. Address = offset + base register. Offset in decimal or hexadecimal.
- Arrays: Base address `0x123B4780`, array[0], 5 elements. Access array[i] by adding `4 * i` to base address.
- Function Calls: Caller invokes callee, passes args, jumps. Callee performs function, returns result, jumps back, restores registers.
- Stack: Stores temp vars, LIFO, grows from high to low memory addresses. sp points to top. Preserved registers saved on stack (s0-s11, ra, sp). Temp registers not saved (t0-t6, a0-a7). Stores return address for multiple/nested functions.
- Control Structures: if, if/else, while achieved with branching instructions
- Interpreting Machine Code: Convert to binary, start with op and funct3.
- Each hexadecimal digit maps to a 4-bit binary number: 0 is 0000, 1 is 0001, 2 is 0010, 3 is 0011, 4 is 0100, 5 is 0101, 6 is 0110, 7 is 0111, 8 is 1000, 9 is 1001, A is 1010, B is 1011, C is 1100, D is 1101, E is 1110, and F is 1111.
- Compressed instructions: 16-bit, use prefix c. `c.addi s0, 4` = `addi s0, s0, 4`

RISC-V Single-Cycle Datapath

- Datapath Components: PC (program counter, holds next instruction address), Register file (32 registers, read/write), Instruction memory (read-only, holds instructions), Data memory (read-write,

holds data), ALU (performs arithmetic/logic operations), Control unit (interprets instruction, generates control signals).
- Supports arithmetic-logical instructions (add, sub, and, or, slt), memory access instructions (lw, sw), control flow instructions (beq).
- Each clock cycle: Fetch instruction from instruction memory at PC, execute instruction (read operands, do ALU operation, write result), update PC.
- Control unit: Reads instruction, decodes, generates control signals, uses ALU zero flag for PC register source.
- Execution of R-Type instruction: Read current PC, increment PC, index into instruction memory, read source operands, perform ALU operation, write result.
- Execution of `lw`: Read current PC, increment PC, index into instruction memory, read base address, read offset and sign extend, add base address and offset, index into data memory, read data, write data.
- Execution of `beq`: Read current PC, increment PC, index into instruction memory, read source operands, compare operands, determine if PC should be updated, target is PC + Extended(imm[12:0], 1'b0).
- Performance: Execution time = `{number of instructions} * {clock cycles per instruction} * {clock cycle time}`. Critical path determines clock cycle time. Slowest instruction (`lw`) determines clock cycles per instruction.

MISC

- Stuck-at-0 faults:
  - RegWrite: R-Type and `lw` malfunction, can't write to register file.
  - ImmSrc1: `lw`, `sw`, `beq` malfunction, upper bit of immediate value always 0.
  - ImmSrc0: `lw`, `sw`, `beq` malfunction, lower bit of immediate value always 0.
  - ResultSrc: `lw` malfunctions, can't write data memory output to register file.
  - ALUSrc: `lw`, `sw`, `beq` malfunction, can't use immediate value as ALU second operand.
- Stuck-at-1 faults:
  - RegWrite: `sw` and `beq` malfunction, write to register file when not supposed to.
  - ImmSrc1: `lw`, `sw`, `beq` malfunction, upper bit of immediate value always 1.
  - ImmSrc0: `lw`, `sw`, `beq` malfunction, lower bit of immediate value always 1.
  - ResultSrc: R-Type malfunctions, can't write ALU result to register file.
  - ALUSrc: R-Type malfunctions, can't use register file value as ALU second operand.

To support the `addi` instruction, we need to make a few modifications to the single-cycle processor datapath:

1. Add a sign-extend block to the datapath:
   - Input: 12-bit immediate value from the instruction
   - Output: 32-bit sign-extended immediate value
2. Modify the ALU source multiplexer:
   - Inputs: Second register operand, sign-extended immediate value
   - Control signal: ALUSrcImm
   - Output: Selected operand for the ALU

3. Update the control signals table for the addi instruction:
    - ALUSrc: 0
    - ALUSrcImm: 1
    - ALUOp: 00
    - MemRead: 0
    - MemWrite: 0
    - MemtoReg: 0
    - RegWrite: 1

To support the jal instruction, we need to make a few modifications to the single-cycle processor datapath:

1. Add a block to calculate the jump target address:
    - Inputs: PC, 20-bit immediate value from the instruction
    - Outputs: Jump target address
2. Modify the PC source multiplexer:
    - Inputs: Incremented PC, jump target address
    - Control signal: PCSrc
    - Output: Selected PC
3. Modify the register write data multiplexer:
    - Inputs: ALU result, incremented PC
    - Control signal: RegDataSrc
    - Output: Selected data for the register file
4. Update the control signals table for the jal instruction:
    - PCSrc: 1
    - RegDataSrc: 1
    - ALUOp: xx
    - MemRead: 0
    - MemWrite: 0
    - MemtoReg: x
    - RegWrite: 1