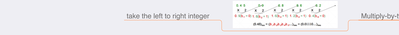
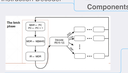
1 Numbers and Representations

- **Properties**
  - leftmost/rightmost digit
  - most/least significant digit
- **Positional Number Systems**
  - Properties
    - range
      - 4-digit r based
    - expand/contract
      - Different Forms, Same Number
  - Example systems
    - Decimal System
    - Hexadecimal System
      - 4 binary to one nibble
    - Binary
  - Conversion
    - Conversion to Decimal
      - $X = \sum_i (a_i)_{ten} \times r_{ten}^{\,i}$
    - Decimal to Binary
      - Division-by-two method
        - take the right to left remains
        -

    - Decimal Fraction to Binary
      - Multiply-by-two method
        - take the left to right integer
        - 
- **Integers in Computers, signed/unsigned**
  - Signed integers
    - C, C++, Java
    - can be positive or negative
    - Negative representation
      - One's Complement
        - all change: 0000, 1111
        - two zeros
        - need an extra step to subtract a number
      - Two's Complement
        - Properties
          - Dfn: N-bit then the complement with respect to 2^N
          - 32-bit word range: -2^31 ~ 2^31-1
            - one negative number with no positive version
          - MSB indicates the sign
            - but MSB is not a sign bit, it is more than sign bit
          - treat 0 as positive, only one 0
        - Methods
          - 5: 0101; -5 = 16-5 = 10000-0101 = 1011
          - shortcut: change all and +1: 0001, 1111
        - Overflow Rule
          - if two numbers with the same sign are added, overflow iff the results has the opposite sign
        - Two's complement integer to decimal
          - positive: just sum
          - negative: -2^(n-1)+sum
  - Unsigned
    - C, C++
    - are used for addresses
    - 32-bit: 0~2^32-1
- **Real Numbers Representation**
  - I + radix point + F
  - Idea: float the radix point
    - standard format for Scientific notation
    - $x = (-1)^S \times (1+\text{Fraction}) \times 2^{(\text{Exponent}-\text{Bias})}$
  - common
    - S: sign bit
    - Exponent
      - unsigned number
      - actual exponent value = Exponent - bias
    - Special cases
      - Expo = 00..00; Fraction = 00.00 — 0_(ten)
      - Expo = 11.11; Fraction = 00.00 — Infinity
      - Expo = 11.11; Fraction != 0.0 — NaN
  - IEEE Standard
    - Single Precision 32-bit
      - S 1 I Exponent 8 I Fraction 23
        - bias: 127
        - range of exponent value: -127~128 (E: 0~255)
        - Exponent 00000000/11111111 reserved
        - range
          - Smallest
            - +- 1.0*2^(-126)
            - Exp = 00000001 = 1-127
            - Fraction = 0..0
          - Largest
            - +- 2.0 * 2^127
            - Exp = 11111110
            - Fraction = 1..1
      - SP to dec
      - binary to SP
      - dec to SP
    - Double Precision 64-bit
      - S 1 I Exponent 11 I Fraction 52
        - bias: 1023
        - range
          - Smallest
            - +- 1.0*2^(-1022)
          - Largest
            - +- 2.0 * 2^1023

2 The von Neumann Model, stored program computer

- von Neumann Machine I stored program computer
  - Basic Idea and Architecture
    - ISA I Instruction Set Architecture > common for all machines
      - interface between hardware and lowest-level software; computer's instructions and their formats
    - RISC philosophy 3.5
      - reduced instruction set computing
    - Microarchitecture > machine dependent
      - the org of the processor, including the major functional units, their interconnection and control
    - layer below ISA, machine dependent
  - Properties
    - instructions and data represented in binary; stored in memory
    - programs can operate on programs
    - binary compatibility
      - standardized ISAs
    - addressable
      - everything has a memory address
      - PC keeps address of instruction being executed
  - Hardware Model
    - Components
      -
    - Processor
      - Control unit, a state machine
        - Functions
          - fetch ins
          - decode ins
          - execute ins
        - Components
          -

            - PC
              - addr of **next** instr to fetch
            - IR I Instruction Register
              - instr fetched from memory
            - Instruction Decoder
              - decode instr and activates necessary circuitry
        - CU state diagram
          - 

        - Mechanism behind: the Clock 2.47
          - Clock generator circuit
            - when CU stops seeing the clock signal, stops processing
          - Stop
            - OS is a special program that manages processor and other resources
          - Method
            - AND the clock generator signal with ZERO
      - ALU I Arithmetic/logic unit
        - Functions
          - mathematical operations
          - logic operations
        - Components
          - circuitry for operations
          - registers
            - store
              - intermediate computational results
            - operands
              - store the result of <=> ops
            - CCR I condition code registers
            - bus for connecting the two
    - Processor-memory interface
    - Memory I RAM
      - properties
        - cell
          - basic unit
          - fixed size
          - each with an address
        - volatile
          - only store when power is on
        - speed
        - expensive
        - a "32-64 bit machine": size of a instruction
        - addr of a memory cell
          - Address width N
            - address space = 2^N
          - size
            - Memory Width W
              - typically 1 byte
            - content of a memory cell
      - MAR I memory address re
      - MDR I memory data re
        - interface
          - Fetch/store signal
          - fetch(address)
          - store(address, value)
          - operations
    - I/O-memory interfaces
    - I/O Units
      - Mass-storage Device Access Methods
        - DASDs I Direct Access Storage Devices
        - SASDs I **Sequential**
          - e.g. tapes
      - slow? I/O Controller
        - Components
          - I/O buffer
          - Control/logic
  - Instruction Processing I Software running mechanism
    - Categories 3
      - Computational
        - ADD
        - AND
      - Data movement
        - LD
        - ST
      - Control
        - JMP, BRnz
        - Op 4 I 000 I Base 3 I Offset 6
    - Properties
      - operation is either executed completely or not at all
    - Phases 6
      - **Fetch instruction from memory**
        - load content of PC into MAR
        - send "read" signal to memory
        - read contents of MDR, store in IR
        - PC+1/2/4..
          - > Changing sequence of instructions
            - jump - unconditional
            - branches - conditional
      - **Decode instr**
        - identify the opcode in the first 4 bits
          - hardware part 4 operation code
        - identify other operands from remaining bits
      - **Evaluate addr**
        - compute addr used for access if the instruction requires memory access
          - add offset to base LDR
          - add offset to PC
          - add offset to zero
      - **Fetch operands from memory**
        - obtain source operands
      - **Execute operation**
        - option 1: send operands to ALU and assert ADD signal
          - option 2: nothing, e.g.load or store
        - perform the operation using the source operands
      - **Store result**
        - write results to destination
          - write address to MAR, data to MDR
          - assert WRITE signal to memory

3 Instruction Set I assembly lang to machine lang in von Neumann machine

- High Level Language Program (e.g., C)
- *Compiler*
- Assembly Language Program (e.g., MIPS)
- *Assembler*
- Machine Language Program (MIPS)
- *Machine Interpretation*
- Hardware Architecture Description (e.g., block diagrams)
- *Architecture Implementation*
- Logic Circuit Description (Circuit Schematic Diagrams)
- computer-org-software

- Assembly Language Program
  - MIPS general, example of RISC instruction set
    - RISC architecture
    - 32-bit instructions
  - **Registers I assembly operands**
    - 32 registers in MIPS
      - each register is 32 bits wide
        - 1 word: 32 bits: 4bytes
    - no type, operation determines how register contents are treated
    - names
      - number references
      - name references
    - Speed
  - Instructions
    - Instr 1-Arithmetic
      - `add`, `sub`
      - immediate operands
        - `addi`, no `subi`
        - 0 is `$zero`
      - Overflow
        - when there is a mistake in arithmetic due to the limited precision in computers
          - unsigned case
          - signed case
        - Handling
    - Instr 2-Data transfer between registers and memory
      - Byte Addressing
        - memory addresses are in **bytes**, not words
        -

      - Byte Order
      - lw $t0, 12($t3)
        - offset must be a constant known at assembly time
        - A[3], 3*4 = 12
      - sw $t0, 40($s3)
        - offset in bytes
      - lb,sb; lbu
        - lbu - unsigned lb
          - filling in the rest with 0
        - lb
          - sign extension
    - Instr 3-Logical
      - logic shifting
        - sll
          - shift 2 bits to the left - multiple 2^2; insert 0's on right
        - srl
          - insert 0's on left
            - only support positive
      - arithmetic shifting
        - sra
          - insert sign bit into left empty
          - arithmetic effect
            - generally divided 2^n
            - especially
              - fails for **odd negative** numbers
                - devision should round **towards** 0
      - and
        - select common 1s, clear others to 0
      - or
        - select all 1s
      - nor
        - A nor 0 = not A
      - xor
        - is 1 when 0-1 or 1-0 (one of two is 1)
    - Instr 4-Control Flow
      - conditional branch
        - beq
        - bne
      - unconditional branch
        - j
        - jr $ra
          - return to addr in $ra, not a fixed place
    - Instr 5-Inequalities
      - slt
        - slt reg1, reg2, reg3
          - reg1 = 1 if yes smaller
      - slti
      - sltu
        - treats registers as unsigned
    - Instr 6-Loop
      - call loop tag
    - Instr 7-Function Calls
      - Structure
        -

      - Registers
        - $a0-a3
        - $v0-v1
        - $ra
      - Function Calls and Return for Leaf function
        - Caller/callee
          - a program can both caller and callee
          - can only be callee, leaf procedure
        - Invoke
          - jal X
            - jump and link
        - Return
          - jr $ra
            - jump register
      - Stack: use more registers
        - Idea
          - except a0-a3 and v0-v1, all registers used by a function call should recover after
        - Stack properties
          - in memory, no need a register
            - $sp
          - convention is grow from **high to low address**
            - push $sp decrease
              - addi $sp, $sp -8
              - sw $t0, 4($sp)
              - sw $a0, 0($sp)
            - pop $sp increase
              - lw $s0, 0($sp)
              - lw $s0, 4($sp)
              - addr $sp, $sp, 8
        - Space allocation
          - automatic variables
            - use stack
          - static variables
            - procedure frame or activation record
              - segment of stack with saved registers and local variables
        - Location in Memory
          - starts in high memory and grows down
            -

          - $gp points to static variables
      - to
      - Function calls and return for Nested procedures
        - Reserve argument and $ra
          - caller save **argument** registers in stack and restore
          - callee save **$ra** in stack and restore
        - other registers to be saved and restored
          - register categories
            - preserved across function call
              - $ra, $sp, $gp, $fp, $s0-7
            - not preserved across function call
              - $v0-1, $a0-3, $t0-9
          - caller
            - do not need to worry about preserved registers
            - for **continue used non preserved registers**, must store and restore
          - callee
            - for **preserved** registers, must store and restore
            - non preserved, nothing
  - Pseudo-instructions
    - move, blt,bgt, bee, bee, li, la
    - TAL \eubset MAL
      - True Assembly Lang vs MIPS AL
- Machine Language Program 32 0/1
  - Idea
    - add $8, $9, $10
    - R format: 00000010101010101010000000100000
      - Machine language instruction
    - hex: 012A 4020
    - dec: 19,546,144
  - Common Props
    - Which format? Look at opcode
    - 6 5 5 5 5 6
    - Which instruction? `opcode + funct`
      - why divide to two fields, looks like similar
    - Save opcode fields make all instructions look like similar
  - **instruction formats**
    - R-format I all others
      - opcode 6 I rs 5 I rt 5 I rd 5 I shamt 5 I funct 6
        - each field is viewed as a 5- or 6-bit **unsigned** integer, not part of a 32-bit integer > 0-31 / 0-63
          - rs rt can only be 0-31
        - shamt: shift amount
          - set to 0 when not `shift` instructions
          - enough: shift more than 31 is useless for 32-bit word
        - opcode for all R ins are 0
    - I-format I instructions with immediate + lw/sw + beq/bne
      - opcode 6 I rs 5 I rt 5 I immediate 16
        - here rt: register to save the result or other operand
        - immediate: **sign-extended** integer
          - large immediate
            - new instruction `lui` I Load Upper Immediate and `ori`
              - lui $at, 0xABAB
              - ori $at, $at, 0xCDCD
              - add $t0, $t0, $at
                - can't be addi coz it's sign extended. want 0 not 1
        - Branching Instructions, beq and bne, I-Format
          - PC-relative addressing on beq/bne
            - immediate fields: two's complement offset to PC
              - number of instructions to jump
                - word-aligned
                - multiple 4 to the address, bytes -> words
                - immediate specify words, not bytes
                - can reach
                - 2^16 instructions; 2^18 bytes around PC
            - **If we don't take the branch:**
              - PC = PC+4 = **next instruction**
            - **If we do take the branch:**
              - PC = **PC+4** + (immediate*4)
              - address: PC-2^15 – PC + 2^15
    - J-format I j and jal
      - can jump to anywhere in memory
        - 32-bit memory
        - 26-bit immediate in j
        -

      - PC-relative addressing on j instructions
        - 1- add 4 to PC
        - 2-Take from j instructions: 26-bit addresses, 28-bit bytes
        - 3-Take from PC: highest 4 order bits
          - can't do when code > 256MB
            - then use 2 jumps or jr instead