**Instruction Sets & Addressing Modes**

* Readings: 2.1 – 2.14
* **Instruction** – a command to CPU to perform a task
  + E.g. move data, perform arithmetic/logic operations, transfer control
  + Execution cycle of a computer
    - Fetch next instruction (store in IR) → decode instruction → fetch operands (if necessary) → perform operation → store result (if necessary)
  + Load – e.g. load R2, LOC – read contents from address LOC and store in register R2
  + Operations – e.g. ADD R4, R2, R3 – add contents of R2 & R3 and store result in R4
  + Store – e.g. store R4, LOC – copy contents of R4 into address LOC
  + One instruction may require many memory operations
    - E.g. fetching the instruction, fetching operands, storing the result
    - The memory device interface is a bottleneck – must be designed and used efficiently
* **RISC – reduced instruction set computers**
  + Each instruction implements a simple function; e.g. add, sub
  + Simpler for compilers to generate code
  + Each instruction fits in a single word
  + Load/store architecture
    - Memory operands are accessed only using Load/Store
    - All operands needed for arithmetic/logic operations must be already loaded into processor registers or provided explicitly
* **CISC – complex instruction set computers**
  + Each instruction can implement complex functionality; e.g. insertInQueue
  + Difficult for compilers to decipher
  + Instructions have variable sizes
  + ALU can perform memory operationsw
* Instruction lengths depend on the address modes used
* Memory addressing
  + Instructions are not allowed to address individual bits in memory
  + **Address space** – the range of addresses
    - Depends on the width of the address signal; e.g. 24-bit address generates an address space of 224 locations
  + Memory systems are typically byte addressable
    - Each byte has unique address; adjacent words have addresses that differ by the # of bytes in a word
  + Word addressable system
    - Each word has unique address; adjacent words differ by 1
* Word & byte encoding
  + A 32-bit word can store a 32-bit int or 4 bytes (e.g. ASCII chars)
* Byte ordering
  + **Big-endian** – assigns lower byte addresses to more significant bytes
    - i.e. 0 address starts at MSB (left → right)
  + **Little-endian** – assigns lower byte addresses to less significant bytes
    - i.e. 0 address starts at LSB (right → left)
* **Aligned address** – begins at a byte address that is a multiple of the number of bytes in a word
  + E.g. a 4-byte word has aligned addresses at 0x0, 0x4, 0x8, 0xc, …
* **Instruction types**
* A computer must have instructions capable of performing 4 types of operations:
  + Data transfer b/t memory and processor registers
  + Arithmetic & logic operations
  + Program sequencing & control
  + I/O transfers (typically memory mapped; same as data transfer operations)
* **Instruction-set architecture (ISA)** – defines set of opcodes (machine language) and native command implemented by a processor
* **Register transfer level (RTL) notation**
  + Expresses semantics of instructions as data transfers
  + Memory locations – e.g. LOC1, LOC2
  + Registers – e.g. R0, R1
  + Contents of register/memory pointed by address – e.g. [R0], [LOC3]
    - Similar to dereferencing a pointer
  + Control signals – e.g. T1, T2
  + Assignment statements
    - Destination ← value
    - E.g. R1 ← [LOC1] – assign value at LOC1 to R1
    - E.g. R3 ← [LOC1] + [LOC2]
* **Addressing modes**
  + Location of instruction operands are specified using effective address (EA)
  + Effect address is computed differently depending on different address modes
  + Immediate addressing – operand is a constant
    - #X → operand = X
  + Register addressing – operand is a register
    - Ri → EA = Ri
  + Absolute/direct addressing – operand address in memory is known
    - LOC → EA = LOC
  + E.g. load R1, B → R1 – register addressing; B – absolute addressing
  + Indirect addressing – operand address is stored in a register/memory location
    - Analogous to a pointer
    - (Ri) → EA = [Ri]
  + E.g. load R0, (R1) → load contents of memory location stored by R1 into R0
    - RTL notation: R0 ← [[R1]]
  + **Move R1, #NUM** – RTL: R1 ← NUM
  + E.g. summing N numbers in memory
    - N, SUM, NUM1, NUM2 … are labels in memory
    - Load R2, N load # of numbers into counter
    - Clear R3
    - Move R4, #NUM1 put address of first number into R4
    - LOOP: load R5, (R4) load value of first number (indirect addressing)
    - Add R3, R3, R5 add sum
    - Add R4, R4, #4 increment memory counter to next word
    - Subtract R2, R2, #1 decrement counter
    - Branch\_if\_R2>0 LOOP return to LOOP label if counter > 0
    - Store R3, SUM
  + Index addressing – operand address is at an offset from a known memory location
    - Useful for arrays/list data structures
    - X(Ri) → EA = [Ri] + X
    - E.g. load R1, 20(R0) – RTL: R1 ← [[R0] + 20]
    - The offset can be X or it can be stored in Ri
  + Base with index addressing – operand address’s offset from a known memory location is stored in a second register
    - Useful for 2D arrays
    - (Ri, Rj) → EA = [Ri] + [Rj]
    - E.g. load R2, (R0, R1) – RTL: R2 ← [[R0] + [R1]]
  + Base with index & offset addressing – operand address is computed from 2 register values and a constant offset
    - Useful for 3D arrays
    - X(Ri, Rj) → EA = [Ri] + [Rj] + X
  + PC-relative addressing – similar to index addressing, except the PC register is used
    - Useful for specifying destinations for branch instructions
    - X(PC) → EA = [PC] + X
  + Auto-increment addressing – simplifies access of successive locations in memory
    - (Ri)+ → EA = [Ri]; [Ri] + increment
    - Register value is incremented after it is read
    - Size of increment depends on the # of bytes read
  + Auto-decrement addressing
    - -(Ri) → [Ri] – decrement; EA = [Ri]
    - Register value is decremented before it is read
* **Assembly language**
  + Can specify one or more source operands & one or more destination operands
  + E.g. **load R1, LOC1** – RTL: R1 ← [LOC1]
  + E.g. **store R1, LOC1** – RTL: LOC1 ← [R1]
  + E.g. **add R3, R1, R2** – RTL: R3 ← [R1] + [R2]
  + Branching
    - E.g. **branch\_if\_[R2]>0 LOOP**
      * If [R2] > 0, PC ← LOOP (a label for an instruction address)
      * Initialize R2 ← # of iterations
      * Sub R2, R2, #1 every iteration of loop