**LEC 2**

* **Register** – small amount of fast memory on CPU (as opposed to RAM)
  + 32 general purpose registers called $0 … $31
  + Each holds 32 bits
  + CPU can only operate on data in registers
  + $0 is always 0
  + Example reg. operation – “add the contents of $s & $t, put the result in $d”
* **RAM** – large amount of memory away from CPU
  + Data travels b/t CPU and RAM on the bus
  + Big array of ~109 bytes
  + Each cell has an address – 0, 1, … n – 1
  + Each 4-byte block is a word
  + Word addresses are 0, 4, 8, c, 10, 14, 18, 1c, … (count in hex)
  + RAM access is much slower than register access
* Communicating w/ RAM:
  + **Load** – transfer a word from an address to a reg.
    - Desired address → **memory access register (MAR)**
      * Goes on the bus, retrieves data at that location
      * Data comes back on the bus and → **memory data register (MDR)**
      * Value in MDR moved to destination register
  + **Store** – does the reverse of load
* The computer doesn’t know which words contain instructions & which contain data
  + **Program counter register (PC)** holds the address of the next instruction to execute (while the current instruction is executing)
  + By convention, guarantee that some address (e.g. 0) contains code, and initialize PC to 0
  + Computer then runs the **fetch-execute cycle**
    - Instruction register (IR) holds the current instruction

PC ← 0 (initialize PC)

Loop

IR ← MEM[PC]

PC ← PC + 4 (increment to next word)

Decode & execute instruction in IR

End loop

* + - The only program the machine runs
* How does a program get executed?
  + **Loader** is a program that puts the program in memory & initializes PC to the address of the first instruction in the program
* What happens when a program ends?
  + Must return control to the loader
    - Set PC to address of the next instruction in the loader ← $31 contains this address
    - E.g. jump register instruction: jr $31 → sets pc = $31
* **Assembly language**
  + Replace tedious binary/hex encodings with easier-to-read mnemonics
    - Less chance of error
    - Translation to binary can be automated (i.e. assembler)
  + One line of assembly = one machine instruction
* Example: add value in $5 to value in $7, store result in $3, and return

|  |  |  |  |
| --- | --- | --- | --- |
| Location | Binary | Hex | Meaning |
| 00000000 |  |  | add $3, $5, $7 |
| 00000004 |  |  | jr $31 |

* Example: add 42 to 52, store in $3, return
  + **lis $d – load immediate & skip**
    - Treat the next word as an immediate value (constant) and load it into $d, then skip to next instruction
  + **.word i** 
    - Not an instruction; a directive that says the next word in the binary should be i
  + lis $5
  + .word 42 – load the constant 42 into $5
  + lis $7
  + .word 52 – load the constant 52 in $7
  + add $3, $5, $7
  + jr $31
* Example: compute the absolute value of $1, store in $1, return
  + **beq $s, $t, i – branch on equal**
    - Increment PC by i words if $s and $t have equal contents
  + **slt $d, $s, $t – set less than**
    - $d is set to 1 if $s < $t, 0 if otherwise
  + slt $2, $1, $0 – compare if $1 < 0 ($0 is always 0)
  + beq $2, $0, 1 – if above returned false ($2 == 0), skip one step
  + sub $1, $0, $1 – subtract from 0 to negate $1
  + jr $31
* Example: sum the integers 1 … 13, store in $3
  + lis $2
  + .word 13 – $2 ← 13 (loop counter)
  + add $3, $0, $0 – $3 ← 0 (initialize sum)
  + add $3, $3, $2 – increment $3 by $2 (add sum) ← loop
  + lis $1
  + .word 1 – $1 ← 1 (loop decrement)
    - Note that this can be done outside of loop; but adding/removing instructions requires updating branch offsets
  + sub $2, $2, $1 – decrement $2
  + bne $2, $0, -5 – if $2 != 0, skip back 5 steps (to 4th step) → loop
    - Note at the end of this step pc already points to jr $31
  + jr $31
  + The assembler allows **labelled instructions**, i.e.
    - label: instruction $s, $t
* Reworked looping:
  + lis $2
  + .word 13
  + add $3, $0, $0
  + top:
  + add $3, $3, $2
  + lis $1
  + .word 1
  + sub $2, $2, $1
  + bne $2, $0, top – assembler calculates the distance b/t pc and top for the branch offset
  + jr $31
* Working with RAM
  + **lw $a, i($b) – load the word at MEM[$b + i] into $a**
  + **sw $a, i($b) – store the word in $a at MEM[$b + i]**
* Ex: $1 = address of array; $2 = # of elements in array; place an element whose index is in $5 into $3
  + **mult $a, $b – multiply**
    - product of 2 32-bit #s is up to 64 bits
    - special registers hi & lo store the result of mult
  + **mflo/mfhi $d – move from lo/hi; store value to $d**
    - For division, lo stores quotient; hi stores remainder
  + lis $4
  + .word 4 – $4 ← 4 for word ←→ byte conversion
  + mult $5, $4
  + mflo $5 – # of bytes to offset
  + add $5, $5, $1 – $5 now holds exact address to fetch from
  + lw $3, 0($5)
  + jr $31