Fall CS 145 Final Review Guide.
֍ Prof: Gordon Cormack
֍ ISA: Ashish Mahto
֍ David Duan - j32duan@edu.uwaterloo.ca
==
Topic. 8: Gen, RAM, CPU.
Subtopic. ★★☆ 8.3 CPU

Note. File from class: cpu.rkt - Internal CPU helper functions and RAM loader.

```
<rkt>
  #lang racket
  (require "RAM.rkt")

  ;; Internal CPU helper functions

  ;; [target-addr] += 1
  (define (cpu-add1 ram target-addr)
    (let* {[old-val (ram-fetch ram target-addr)]
           [new-val (add1 old-val)]
           [new-ram (ram-store ram target-addr new-val)]}
           new-ram))

  ;; [target-addr] += [source-addr]
  (define (cpu-add ram target-addr source-addr)
    (let* {[source-val (ram-fetch ram source-addr)]
           [old-val (ram-fetch ram target-addr)]
           [new-val (+ old-val source-val)]
           [new-ram (ram-store ram target-addr new-val)]}
           new-ram))

  ;; [target-addr] -= [source-addr]
  (define (cpu-sub ram target-addr source-addr)
    (let* {[source-val (ram-fetch ram source-addr)]
           [old-val (ram-fetch ram target-addr)]
           [new-val (- old-val source-val)]
           [new-ram (ram-store ram target-addr new-val)]}
           new-ram))

  ;; if [source-addr] = 0 then [target-addr] += 1
  (define (cpu-add1ifzero ram target-addr source-addr)
    (let* {[source-val (ram-fetch ram source-addr)]}
           (if (zero? source-val)
               (cpu-add1 ram target-addr)
               ram)))

  ;; [target-addr] = [source-addr]
  (define (cpu-copy ram target-addr source-addr)
    (let* {[source-val (ram-fetch ram source-addr)]
           [new-ram (ram-store target-addr source-val)]}
           new-ram))

  ;; cpu-copy takes in two addresses
  ;; target-addr: the address we will store the value
```

```
  ;; source-addr: the address we get our value
  ;; source-pointer-addr contains an address, which is where we want to copy the value from.
  ;; In other words. [source-pointer-addr] = source-addr
  ;; And to get the value stored at source-addr, we need to use [source-addr].
  ;; Hence [[source-pointer-addr]] = [source-addr] = value-at-source-addr
  (define (cpu-fetch ram target-addr source-pointer-addr)
    (let* {[source-addr (ram-fetch ram source-pointer addr)]
           [new-ram (cpu-copy ram target-addr source-addr)]}
           new-ram))

  ;; We want to store the value from source-addr to the address stored inside target-ptr-addr.
  ;; That is, [source-addr] (value from source-addr) --store at--> [target-ptr-addr]
  ;; Aka [target-addr] (value at target-addr) = [[target-ptr-addr]] = [source-addr].
  ;; [[target-ptr-addr]] = [source-addr]
  (define (cpu-store ram target-ptr-addr source-addr)
    (let* {[target-addr (ram-fetch ram target-ptr-addr)]
           [new-ram (cpu-copy ram target-addr source-addr)]}
           new-ram))

  ;; output = [source-addr]
  (define (cpu-output ram source-addr)
    (let* {[output (ram-fetch ram source-addr)]}
      (display output) (newline)
      ram)))

  ;; [target-addr] = input
  (define (cpu-input ram target-addr)
    (ram-store ram target-addr (read)))  ;; (read) takes in value from external sources.


  ;; Loader -- put program and data into RAM

  ;; Put (first lst) at load-addr, (second lst) at (add1 load-addr), and so on.
  (define (ram-load ram load-addr lst)
    (cond
      [(empty? lst) ram]
      [else
        (let* {[value (car lst)]
               [new-list (cdr lst)]
               [current-addr load-addr]
               [new-addr (add1 load-addr)]}
               (ram-load (ram-store ram current-addr value) new-addr new-list))]))

</rkt>
```

Note. Format of machine-language program

1. Say we have a series of instructions [a][a+1][a+2][a+3]···, where a is load-addr.
2. [a]: the first machine instruction is at address load-addr+[a]. In other words, [a] tells us where the real program starts.
3. [a+1] ··· [a+[a]-1]: programmer-defined instructions.
4. [a+[a]] ··· : the rest of the instructions are for the program.

Example. Let a = 0, and [a] = 5.

1. [a] = [0] = 5 tells us that the program starts at addr=5.
2. [a+1] ⋯ [a+[a]-1] = [1] ⋯ [4]: programmer-defined instructions.
3. [a+[a]] ⋯ = [5] ⋯: the rest of the instructions are for the program.

Note. Instruction word

1. Each instruction word is a 5-digit number, ottss, which can be break down into three parts (kappa...)
   o: operation code
   tt: target-addr
   ss: source-addr

2. operation code

0ttss	[tt] += 1
1ttss	[tt] += [ss]
2ttss	[tt] -= [ss]
3ttss	[tt] = [ss]
4ttss	if [ss] = 0 then [tt] += 1
5ttss	[tt] = [[ss]]
6ttss	[[tt]] = [ss]
7ttss	display [ss]
8ttss	read [tt]
9ttss	halt (stop program flow)

Note. CPU fetch-eval cycle

```
<rkt>
  (define (cpu ram instruction-ptr-addr)
    (define (cycle ram)

      ;; [instruction-ptr-addr] tells us where to get our instruction.
      (define instruction-pointer (ram-fetch ram instruction-ptr-addr))

      ;; [instruction-pointer] tells us what the instruction is.
      ;; ie. [[instruction-ptr-addr]] = [instruction-pointer] = instruction.
      (define instruction (ram-fetch ram instruction-pointer))

      ;; Now we want to increment our instruction-ptr-addr by 1, which tells the
      ;; computer what we will be doing in the next iteration.
      ;; Recall that (cpu-add1 ram target-addr) => [target-addr] += 1
      (define r1 (cpu-add1 ram instruction-ptr-addr))

      ;; We want to extract the operation-code by dividing instruction by 10000.
      ;; See above "Note. Instruction word" for further explanation.
      ;; Remark: (quotient instruction 10000) gives us the first digit of instruction.
      (define operation-code (quotient instruction 10000))

      ;; Remark: (modulo (quotient 59601 100) 100) = (modulo 596 100) = 96 = tt
      (define target-addr (modulo (quotient instruction 100) 100))

      ;; Remark: (modulo (59601 100)) = 01 = ss
      (define source-addr (modulo instruction 100))

      ;; Now we define the instructions. See above "Note. Format of machine-language
```

```
      program" for further explanation.
      (let* {[op operation-code]} ;; sorry I'm too lazy to type operation-code xd
        (cond
          [(= 0 op) (cycle (cpu-add1 r1 target-addr))]
          [(= 1 op) (cycle (cpu-add r1 target-addr source-addr))]
          [(= 2 op) (cycle (cpu-sub r1 target-addr source-addr))]
          [(= 3 op) (cycle (cpu-copy r1 target-addr source-addr))]
          [(= 4 op) (cycle (cpu-add1ifzero r1 target-addr source-addr))]
          [(= 5 op) (cycle (cpu-fetch r1 target-addr source-addr)]
          [(= 6 op) (cycle (cpu-store r1 target-addr source-addr))]
          [(= 7 op) (cycle (cpu-output r1 source-addr))]
          [(= 8 op) (cycle (cpu-input r1 target-addr))]
          [(= 9 op) r1])))) ;; we stop calling (cycle instruction).
    (cycle ram))

  </rkt>
```

Example. Sample Machine Language program
Outputs 84, 42, then echoes (numeric) input until 0 is input.

```
<rkt>
  (define r1 (ram-load ram 0 ;; load-addr = 0
    '( 2      ;; [0] = 2 : first instruction starts at address 2.
      42      ;; [1] = 42: programmer-defined literal, not an instruction.
      60101   ;; op = 6:  (cpu-store r1 target-addr source-addr)
                          [[tt]] = [ss]  => [[01]] = [01] = 42
                                         => [42] = 42
                          ie. We store number 42 at address 42.
      10101   ;; op = 1:  (cpu-add r1 target-addr source-addr)
                          [tt] += [ss] => [1] += [1]  => 42 += 42
                                                      => [1] = 84
                          ie. We store number 84 at address 1.
      70001   ;; op = 7:  (cpu-output r1 source-addr)
                          Output [1] => display [1] => display 84
      70042   ;; op = 7:  (cpu-output r1 source-addr)
                          Output [42] => display [42] => display 42
      34400   ;; op = 3:  (cycle (cpu-copy r1 target-addr source-addr))
                          [tt] = [ss] => [44] = [00] => [44] = 02
                          ie. We save the instruction pointer at address 44
                              for later use. Right now [44] points to our
                              next instruction 85500. This allows us to do
                              a loop (see lines 40055 and 30044).
      85500   ;; op = 8:  (cycle (cpu-input r1 target-addr))
                          read [tt] => read 55 => store user-input at addr 55.
                          Remark: 00 is garbage value in this case.
                          ie. Now at address 55 we have a user-input value.
      70055   ;; op = 7:  (cycle (cpu-output r1 source-addr))
                          out [tt] => out [55] => output user-input value.
      40055   ;; op = 4:  (cycle (cpu-add1ifzero r1 target-addr source-addr))
                          if [ss] = 0, [tt] += 1 => if [55] = 0, [00] += 1
                          ie. if input = 0, increment instruction-pointer, which
                              would allow us to skip next line (skipping the loop)
                              and jump directly to the end. Otherwise, go to the
                              next line, which would bring us into a loop.
```

```
       30044  ;; op = 3:  (cycle (cpu-copy r1 target-addr source-addr))
                          [tt] = [ss] => [00] = [44]
                          ie. We copy the value stored at address 44 to our
                              instruction pointer, thus creating a loop.
       90000  ;; op = 9:  We reach this line only when the line 40055 skips 30044,
                          or that [55] = 0. 90000 ends our program.
       )))

  (cpu r1 0)
</rkt>
```

Example. Machine language programs to sum numbers from 1 to n

```
<rkt>
  (define r (cpu (ram-load ram 0
                   '(6     ; [0]: first instruction
                     1     ; [1]: literal 1
                     2     ; [2]: literal 2
                     7     ; [3]: literal 7
                     0     ; [4]: n
                     0     ; [5]: acc
                     80400 ; input n
                     40004 ; skip next if n = 0
                     10002 ; skip next 2 instructions
                     70005 ; out acc
                     90000 ; halt
                     10504 ; acc = acc + n
                     20401 ; n = n - 1
                     20003 ; go back 7 (from next instr)
                     )) 0))
</rkt>
```

==

END