Quick review

Your core loop and feature set are solid (input→physics→collision→animation). Below are targeted fixes, optimizations, and a few quality-of-life mechanics. I included drop-in snippets you can paste into your cart.

---

1) Bugs & correctness fixes

A. `get_flags()` **right-side Y samples**
In the *right* check you accidentally sample `top_left.y/8`, `mid_left.y/8`, `bot_left.y/8` instead of the **right** points. This can misreport wall flags.

```
-- fix the RIGHT samples
if fget(mget((top_right.x +1)/8, top_right.y/8), flag) or
   fget(mget((mid_right.x+1)/8, mid_right.y/8), flag) or
   fget(mget((bot_right.x+1)/8, bot_right.y/8), flag) then
  env.right = env.right | (1<<flag)
end
```

B. `collision_y()` **sets** `player.on_ground` **instead of** `obj.on_ground`
That makes the function non-reusable and can desync if you ever pass a different object.

```
-- inside collision_y(obj)
if (bottomCollides) then
  obj.on_ground = true
  if obj.dy > 0 then
    obj.dy = math.floor(bot_mid.y/8)*8 - (obj.y + obj.h)
  end
else
  obj.on_ground = false
end
```

C. Wall fall speed:
In `gravity()` the wall state hard-sets `dy = 0.4`. That can override upward motion and feel sticky. Prefer clamping instead of overwriting:

```
if obj.state == 'wall' then
  obj.dy = math.min(obj.dy + GRAVITY*0.6, 1.5)
else
  obj.dy = math.min(obj.dy + GRAVITY, 7)
end
```

D. Animation `loop` is global
`loop` is a global used by `draw_object()`. If you ever draw multiple objects/FX, frames will fight. Store loop direction per object:

```
-- add per-object anim fields
player.anim_loop = 1
player.anim_counter = 0

-- update draw_object(obj) to use obj.anim_loop/anim_counter
```

E. `player_input()` sets `player.craw/attack` booleans but uses `player.action`
These fields are dead code; remove them or standardize on one approach.

F. Horizontal knockback can push into tiles
`check_env()` sets `dx = ±4` before collision; if you're flush to a wall you may get stuck. After applying knockback, run one immediate collision step or clamp dx by available free space (see Sweep snippet below).

---

2) Performance & readability

A. Replace `math.pow(2, flag)` with bit shifts. Lua in TIC-80 supports bitwise ops; you already use `|`/`&`.

```
local MASK = 1<<flag
-- then: env.up = env.up | MASK, and checks with (env.up & MASK) ~= 0
```

B. Avoid tiny table allocations in hot paths
`{x=...,y=...}` is created dozens of times per frame in collision. Inline locals instead:

```
-- example inside collision_x
local tlx = obj.x - 1 + obj.ox + obj.dx
local tly = obj.y
-- ... compute once; reuse
```

C. Tile lookups helper
You call `mget/fget` repeatedly. Make a fast helper that uses shifts:

```
local function tile_flag(px, py, flag)
  return fget(mget(px>>3, py>>3), flag)
end
```

D. Friction/air drag rounding
Repeated `math.floor/ceil(10*dx*FRICTION)/10` creates lots of float ops. Snap small speeds to zero and otherwise multiply once:

```
local function apply_drag(v, factor)
  v = v*factor
  if math.abs(v) < 0.05 then return 0 end
  return v
end
-- in gravity(): obj.dx = apply_drag(obj.dx, obj.on_ground and FRICTION or AIR)
```

---

3) State machine simplification

Your `player_states()` is clear, but you can remove string churn and centralize transitions. Use numeric constants and a table of handlers. Example pattern:

```
ST = {IDLE=1,RUN=2,JUMP=3,FALL=4,WALL=5,CRAW=6,ATTACK=7,DEATH=8}

local function enter(state)
  player.state = state
  player.counter = 0
end

local function want_wall()
  return (env.left & (1<<2) ~= 0 or env.right & (1<<2) ~= 0)
end

local function compute_state()
  if player.hp <= 0 then return ST.DEATH end
  if player.action == 'attack' then return ST.ATTACK end
  if player.action == 'craw'   then return ST.CRAW end
  if player.dy < 0             then return ST.JUMP end
  if not player.on_ground then
    if want_wall() then player.jumps = 1; return ST.WALL end
    return ST.FALL
  end
  if player.dx ~= 0 then return ST.RUN end
  return ST.IDLE
end

-- in update
local ns = compute_state()
```

```
if ns ~= player.state then enter(ns) end
player.flip = (player.dx < 0) and 1 or (player.dx>0 and 0 or player.flip)
```

Benefits: fewer string comparisons, cleaner transitions, and predictable entry hooks.

---

4) Input, coyote time & jump buffer (feel)

Two small helpers dramatically improve platformer feel.

```
player.coyote = 0   -- frames you can still jump after leaving ground
player.buf    = 0   -- frames you can buffer a jump before landing
COYOTE_MAX = 8
BUF_MAX    = 8

local function update_jump_timers()
  if player.on_ground then
    player.coyote = COYOTE_MAX
  elseif player.coyote > 0 then
    player.coyote = player.coyote - 1
  end
  if btnp(0) or btnp(4) then player.buf = BUF_MAX end
  if player.buf > 0 then player.buf = player.buf - 1 end
end

local function try_jump()
  -- allow: on ground via coyote OR remaining double-jumps
  local can_ground = player.coyote > 0
  local can_air    = player.jumps < player.max_jumps
  if player.buf > 0 and (can_ground or can_air) then
    player.buf = 0
    if player.state == 'wall' then
      local dir = (player.flip==0) and -1 or 1 -- push away from wall
      player.dx = dir*(player.speed+1.5)
    end
    player.dy = -player.jump
    if not can_ground then player.jumps = player.jumps + 1 end
  end
end
```

Call `update_jump_timers()` early in `player_update()`, then `try_jump()` after you know `on_ground`/wall.

---

5) Update order (deterministic & stable)

Current order mixes state calc with collision/environment. A tidy order reduces edge cases:

1. **Read inputs** → desired horizontal accel/intent (don't set `dx` directly to `±speed`; use acceleration for smoother control).
2. **Apply timers** (coyote/buffer, damage i-frames).
3. **Apply gravity + friction/drag** (to velocities).
4. **Integrate X**; **resolve X collisions**.
5. **Integrate Y**; **resolve Y collisions** and set `on_ground`.
6. **State compute** (now that `on_ground`, `dy`, wall contact known).
7. **Animation**.

That avoids declaring `player.on_ground` before collisions. You're close—just move `player_states()` after collisions.

---

6) Swept tile collision (micro-sweep)

To prevent tunneling and make knockback safe, sweep one tile at a time along each axis based on the sign of the velocity:

```
local function move_axis(obj, ax, v)
  local step = (v>0) and 1 or -1
  local rem = math.abs(v)
  while rem >= 1 do
    if ax=='x' then obj.x = obj.x + step else obj.y = obj.y + step end
    if (ax=='x' and hits_solid_x(obj)) or (ax=='y' and hits_solid_y(obj)) then
      if ax=='x' then obj.x = obj.x - step; obj.dx = 0 else obj.y = obj.y -
step; obj.dy = 0 end
      return
    end
    rem = rem - 1
  end
  -- fractional remainder
  if rem>0 then
    local frac = step*rem
    if ax=='x' then obj.x = obj.x + frac; if hits_solid_x(obj) then obj.x =
obj.x - frac; obj.dx = 0 end
    else obj.y = obj.y + frac; if hits_solid_y(obj) then obj.y = obj.y - frac;
obj.dy = 0 end end
  end
end
```

Hook this inside `player_update()` instead of `player.x = player.x + player.dx` / `player.y = player.y + player.dy`.

---

7) Environment flags

You recompute all 8 flags every frame even if you only need a couple. If flag `0` is "solid" and `1` is "hazard", build helpers:

```
local SOLID, HAZ = 0, 1
local function solid_at(px,py) return fget(mget(px>>3,py>>3), SOLID) end
local function hazard_at(px,py) return fget(mget(px>>3,py>>3), HAZ) end
```

Then probe only what you need in collisions and damage. It's faster and simpler to reason about than accumulating bitmasks in `env`.

If you keep the bitmask approach, convert to shifts: `(env.up & (1<<HAZ)) ~= 0`.

---

8) Knockback & i-frames polish

Use a short grace where the player can't be *re-launched* by hazards but still collides with solids; also add a flicker:

```
player.inv = 0
local function take_damage(obj, kx, ky)
  if obj.inv > 0 then return end
  obj.hp = obj.hp - 1
  obj.dx, obj.dy = kx, ky
  obj.inv = 60
end

-- in update
if player.inv>0 then player.inv = player.inv - 1 end

-- in draw
local hidden = (player.inv>0) and ((t//2)%2==0)
if not hidden then draw_object(player) end
```

---

9) Camera

Currently you clamp with magic numbers. Prefer deriving from map size if you use multiple rooms:

```
-- map width/height in tiles
local MW, MH = 240, 136  -- TIC-80 full map; or pass room bounds
local SW, SH = 240, 136
camera.x = math.max(0, math.min(player.x - SW//2, MW*8 - SW))
```

For parallax or look-ahead, offset by `player.dx*8` when running.

---

10) Misc micro-tweaks

- Cache `btn()` / `btnp()` results once per frame.
- Precompute `player.ox_left = player.ox` and `player.ox_right = player.w-1-player.ox` to avoid recomputing both expressions everywhere.
- Replace `map(map_x, map_y, 31, 18, ...)` hardcoded numbers with screen-derived constants.
- Consider integer positions for tiles but keep velocities as floats; you already do this—good.

---

11) Drop-in: acceleration-based horizontal control

```
-- params
a_ccel = 0.6   -- ground accel
a_air  = 0.35  -- air accel
max_sp = 2.6

local function horiz_control()
  local left  = btn(2)
  local right = btn(3)
  local a = player.on_ground and a_ccel or a_air
  local intent = (right and 1 or 0) - (left and 1 or 0)
  player.dx = player.dx + intent*a
  if intent==0 and player.on_ground then
    player.dx = apply_drag(player.dx, FRICTION)
  end
  if player.dx> max_sp then player.dx = max_sp elseif player.dx< -max_sp then
player.dx = -max_sp end
end
```

Call `horiz_control()` instead of writing `dx = ±speed`.

---

12) Animation selection table (clean)

```
local ANIM = {
  [ST.IDLE]=idle_animation,
  [ST.RUN]=run_animation,
  [ST.JUMP]=jump_animation,
  [ST.FALL]=fall_animation,
  [ST.CRAW]=craw_animation,
  [ST.WALL]=wall_animation,
  [ST.ATTACK]=attack_animation,
  [ST.DEATH]=death_animation,
}

player.anim = ANIM[player.state]
```

---

13) What to measure

- Frame time with/without the new collision helpers (profile by counting mget/fget per frame).
- False wall detections after the right-sample fix.
- Feel improvements from coyote/buffer (missed jumps should drop dramatically).

---

14) Nice-to-have mechanics (later)

- Variable jump height (cut upward velocity when jump released early).
- Ledge forgiveness when rising into a tile (ignore a single-frame head bonk).
- One-way platforms (check only when `dy>0`).

If you want, paste your updated `player_update()` after trying the re-order and I'll do a second pass for edge cases.