Super Mario Bros - Assembly Language Game Documentation

Roll Number: 24I-0624

Table of Contents

Game Overview

This is a full-featured Super Mario Bros clone written entirely in x86 Assembly Language using the Irvine32 library. The game implements:

- **2 Complete Levels** with unique layouts
- **Multiple Enemy Types**: Goombas, Koopas, and Shells
- **Power-up System**: Small Mario → Big Mario → Fire Mario
- **Advanced Physics**: Gravity, jumping, double-jumping
- **Sound Effects**: MP3 playback via Windows MCI (Media Control Interface)
- **Save System**: High scores (top 3) and game progress
- **Special Features**: Freeze power-up, mushrooms, breakable platforms, mystery blocks
- **Time-based Scoring**: 150-second timer per level with bonus points

The game runs at approximately 30 FPS (45ms delay per frame) and features a 120x27 tile-based map with scrolling viewport.

Architecture & Control Flow

The Main Game Loop

The heart of the entire game is the `main` procedure. Think of it as the master conductor of an orchestra - it initializes everything and then continuously checks what state the game is in, delegating work to the appropriate "section" (procedure).

```
main PROC
    call Randomize          ; Seed random number generator
    call Clrscr            ; Clear the screen
    call HideCursor        ; Hide that blinking cursor
    call InitSounds        ; Load MP3 files via MCI
    call LoadScore         ; Load top 3 high scores from file
    call LoadProgress      ; Load saved game progress

MainLoop:
    ; This is the eternal loop - it only exits when gameState = 255
    mov al, gameState      ; Load current state into AL
    cmp al, 0              ; Is it 0? Menu screen
    je DoMenu
    cmp al, 1              ; Is it 1? Playing the game
    je DoPlay
    cmp al, 2              ; Is it 2? Paused
    je DoPause
    cmp al, 3              ; Is it 3? Level complete
    je DoLvlDone
    cmp al, 4              ; Is it 4? Game over
    je DoOver
    cmp al, 5              ; Is it 5? Victory (beat game)
    je DoWin
    jmp ExitProg           ; Any other value = exit

DoMenu:
    call MenuScreen
    jmp MainLoop           ; Return to loop after menu
DoPlay:
    call PlayGame
    jmp MainLoop
; ... (similar for other states)

ExitProg:
    call SaveScore         ; Save before exiting
    call SaveProgress
    call ShowCursor        ; Restore cursor
    exit                   ; Exit to DOS/Windows
```

```
main ENDP
```

Why This Design?

The game uses a **state machine pattern**. The `gameState` byte acts as a controller - by changing its value, any procedure can instantly switch what the game is doing. For example, when you press 'P' during gameplay, the input handler simply sets `gameState = 2`, and on the next iteration of MainLoop, the game switches to the pause screen. Clean and simple!

Data Structures

Constants

The game defines several critical constants that control the entire game world:

```
SCREEN_W = 120          ; Screen width in characters
SCREEN_H = 30           ; Screen height
MAP_W = 120             ; Map width (same as screen - no scrolling)
MAP_H = 27              ; Map height
MAX_ENEMIES = 10        ; Maximum enemies alive at once
MAX_COINS = 20          ; Maximum coins in a level
MAX_FIREBALLS = 5       ; Maximum fireballs Mario can have active

; Physics constants
JUMP_VELOCITY = -4      ; Initial upward velocity when jumping (negative = up)
DOUBLE_JUMP_VEL = -3    ; Slightly weaker double jump
GRAVITY = 1             ; Added to velocity each frame
MAX_FALL_SPEED = 4      ; Terminal velocity

; Score requirements
SCORE_L1 = 700          ; Need 700+ score to pass level 1
SCORE_L2 = 500          ; Need 500+ score to pass level 2
```

Why Negative Jump Velocity? In our coordinate system, Y increases downward (Y=0 is top of screen). So to move UP, we need to decrease Y, which means negative velocity. This is standard in most 2D games.

Mario's State

Mario's entire existence is tracked by these variables:

```
marioX BYTE 5               ; Current X position (horizontal)
marioY BYTE 24              ; Current Y position (vertical)
prevMarioX BYTE 5           ; Previous X (for erasing old sprite)
prevMarioY BYTE 24          ; Previous Y
marioVelY SBYTE 0           ; Vertical velocity (SIGNED - can be negative!)
```

```
marioLives BYTE 3          ; Number of lives remaining
marioPower BYTE 0          ; Power state: 0=small, 1=big, 2=fire
marioOnGround BYTE 1       ; Boolean: 1=on ground, 0=in air
marioInvinc BYTE 0         ; Boolean: 1=invincible (just got hit)
invincTimer WORD 0         ; Frames of invincibility remaining
canDoubleJump BYTE 1       ; Can Mario double jump?
usedDouble BYTE 0          ; Has double jump been used?
marioDir BYTE 1            ; Direction: 0=left, 1=right
```

Why SBYTE for Velocity? Velocity can be positive (falling down) or negative (jumping up). Regular BYTE is unsigned (0-255), but SBYTE is signed (-128 to +127), allowing negative values!

Enemy System

Enemies are stored in **parallel arrays** - each index represents one enemy:

```
enemyX BYTE MAX_ENEMIES DUP(0)        ; X positions
enemyY BYTE MAX_ENEMIES DUP(0)        ; Y positions
enemyPrevX BYTE MAX_ENEMIES DUP(0)    ; Previous X (for erasing)
enemyPrevY BYTE MAX_ENEMIES DUP(0)    ; Previous Y
enemyDir BYTE MAX_ENEMIES DUP(0)      ; Direction: 0=left, 1=right
enemyActive BYTE MAX_ENEMIES DUP(0)   ; Is this enemy alive?
enemyType BYTE MAX_ENEMIES DUP(0)     ; 0=Goomba, 1=Koopa, 2=Shell
enemyCount BYTE 0                     ; Total enemies in level
```

Array Indexing Example: If we want enemy #3's X position, we access `enemyX[3]`. In assembly, this is: `mov al, enemyX[ebx]` where EBX=3.

The Level Map

The entire level is stored as a **linear array** of 3240 bytes (120 width × 27 height):

```
levelMap BYTE 3240 DUP(' ')           ; The actual map data
```

Each byte represents one tile: - `#` = Brick wall - `=` = Breakable platform - `?` = Mystery block - `-` = Used mystery block - `o` = Coin (removed from map when level loads) - `G` = Goomba spawn (removed from map) - `K` = Koopa spawn (removed from map) - `F` = Flag (goal) - `C` = Freeze power-up - ` ` = Empty space

Converting 2D to 1D: To access tile at (X=10, Y=5):

```
Index = Y × MAP_W + X = 5 × 120 + 10 = 610
Access: levelMap[610]
```

Sound System

InitSounds - Loading MP3 Files

This procedure uses Windows MCI (Media Control Interface) to load MP3 files. MCI is like a universal remote control for multimedia - you send it text commands and it does the work.

```
InitSounds PROC
    pushad                  ; Save all registers (we're being polite!)

    ; Open jump.mp3 and give it the alias "jump"
    INVOKE mciSendStringA, OFFSET sndJump, NULL, 0, 0
    ; The command is: "open jump.mp3 type mpegvideo alias jump"
    ; This tells Windows: "Hey, open this MP3 and call it 'jump' so I can reference it later

    ; Open collectcoin.mp3 with alias "coin"
    INVOKE mciSendStringA, OFFSET sndCoin, NULL, 0, 0

    ; Open shootfire.mp3 with alias "fire"
    INVOKE mciSendStringA, OFFSET sndFire, NULL, 0, 0

    mov soundsLoaded, 1     ; Flag: sounds are ready!

    popad                   ; Restore all registers
    ret
InitSounds ENDP
```

Why Aliases? Instead of saying "play C:tojump.mp3" every time, we just say "play jump". Much cleaner!

PlayJumpSound - Playing with Fallback

```
PlayJumpSound PROC
    pushad                      ; Save registers

    cmp soundsLoaded, 1         ; Did InitSounds succeed?
    jne JumpBeep                ; No? Use backup beep instead

    ; Play the MP3: "play jump from 0"
    ; This means: "Play the sound 'jump' starting from the beginning"
    INVOKE mciSendStringA, OFFSET sndJumpPlay, NULL, 0, 0
    jmp JumpDone

JumpBeep:
    ; Fallback: Use Windows Beep API
    INVOKE Beep, 600, 50        ; Beep(frequency=600Hz, duration=50ms)
    ; 600Hz gives a jump-like "boing" sound
```

```
JumpDone:
    popad
    ret
PlayJumpSound ENDP
```

Why Fallback? If the MP3 files are missing, the game still works - it just uses system beeps. This is **defensive programming**!

Sound Frequency Psychology

Notice the different frequencies: - **Jump**: 600Hz - Medium-high pitch (bouncy feeling) - **Coin**: 1200Hz - High pitch (satisfying "ding!") - **Stomp**: 300Hz - Low pitch (heavy "thud") - **Damage**: 200Hz - Very low (danger/hurt feeling) - **Freeze**: 1500Hz - Very high (magical "sparkle")

These aren't random - they're chosen to create the right **emotional response**!

Tile & Collision System

GetTileAt - The Foundation of Collision

Every collision check in the game ultimately calls this procedure. It's like asking "What's at position (X, Y)?"

```
GetTileAt PROC
    ; Input: BL = X position, BH = Y position
    ; Output: AL = tile character at that position

    push ebx              ; Save EBX (we'll modify it)
    push edx              ; Save EDX

    ; Convert 2D coordinates to 1D array index
    ; Formula: index = Y × MAP_W + X

    movzx eax, bh         ; Move Y into EAX, zero-extending
                          ; Why MOVZX? BH is 8-bit, EAX is 32-bit
                          ; MOVZX fills upper bits with zeros

    imul eax, MAP_W       ; EAX = Y × 120
                          ; IMUL = Integer MULtiply

    movzx edx, bl         ; Move X into EDX, zero-extending
    add eax, edx          ; EAX = Y×120 + X (final index!)

    mov esi, OFFSET levelMap    ; ESI points to start of map array
```

```
    mov al, BYTE PTR [esi + eax]  ; Load the tile character
                                   ; BYTE PTR tells assembler: "treat this as 1 byte"

    pop edx               ; Restore registers
    pop ebx
    ret
GetTileAt ENDP
```

Register Choices: - **BL/BH**: Perfect for X/Y since they're just position bytes (0-119, 0-26) - **AL**: Return value - perfect for a single character - **ESI**: By convention, ESI is a "source index" - perfect for reading from arrays

SetTileAt - Modifying the World

Sometimes we need to *change* the map - like breaking a platform or using a mystery block:

```
SetTileAt PROC
    ; Input: BL = X, BH = Y, CL = new tile character
    ; Output: None (modifies map)

    push eax
    push ebx
    push edx

    ; Same index calculation as GetTileAt
    movzx eax, bh         ; Y into EAX
    imul eax, MAP_W       ; Y × 120
    movzx edx, bl         ; X into EDX
    add eax, edx          ; Index complete

    mov esi, OFFSET levelMap
    mov BYTE PTR [esi + eax], cl   ; Write CL (new character) to map

    pop edx
    pop ebx
    pop eax
    ret
SetTileAt ENDP
```

Why CL? CL is the low byte of ECX. Since we only need to pass 1 byte (a character), CL is perfect. This is assembly convention - use the smallest register that fits your data!

IsSolidTile - Wall Detection

This procedure answers: "Can Mario/enemies walk through this tile?"

```
IsSolidTile PROC
    ; Input: AL = tile character
    ; Output: Zero Flag (ZF) = 1 if solid, 0 if not solid

    cmp al, '#'           ; Is it a brick?
    je IsSolid            ; Yes - jump to solid handler
    cmp al, '['           ; Castle wall left?
    je IsSolid
    cmp al, ']'           ; Castle wall right?
    je IsSolid
    cmp al, '-'           ; Used mystery block?
    je IsSolid
    cmp al, '?'           ; Unused mystery block?
    je IsSolid

    ; If we get here, it's NOT solid
    or al, al             ; Set flags based on AL
                          ; OR AL, AL sets ZF=0 (not solid)
    ret

IsSolid:
    xor al, al            ; Set AL to 0
                          ; XOR AL, AL also sets ZF=1 (solid!)
    ret
IsSolidTile ENDP
```

The Zero Flag Trick: Instead of returning 0/1 in AL, we use the **Zero Flag (ZF)**. Callers can immediately use JZ (Jump if Zero) or JNZ (Jump if Not Zero). This is faster than comparing AL!

Usage Example:

```
mov bl, marioX
mov bh, marioY
call GetTileAt        ; AL = tile at Mario's position
call IsSolidTile      ; Check if solid
jz CantMoveThere      ; If ZF=1 (solid), jump
; ... can move there ...
```

IsBreakableTile - Special Platforms

Breakable platforms (=) are different from solid walls - Mario can break them by hitting from below!

```
IsBreakableTile PROC
    ; Input: AL = tile character
    ; Output: ZF = 1 if breakable, 0 if not
```

```
    cmp al, '='           ; Is it the breakable platform character?
    je IsBreakable        ; Yes!

    or al, al             ; Not breakable - set ZF=0
    ret

IsBreakable:
    xor al, al            ; Breakable - set ZF=1
    ret
IsBreakableTile ENDP
```

IsBlockingTile - Combined Check

This is the **master collision checker** - it checks BOTH solid tiles AND breakable platforms:

```
IsBlockingTile PROC
    ; Input: AL = tile character
    ; Output: ZF = 1 if blocks movement, 0 if can pass through

    push eax              ; Save AL (we'll call other procedures)

    call IsSolidTile      ; Is it solid?
    jz ItsBlocking        ; Yes! ZF=1, so we're done

    ; Not solid, but maybe breakable?
    pop eax               ; Restore AL
    push eax              ; Save again

    call IsBreakableTile  ; Is it breakable?
    jz ItsBlocking        ; Yes! That also blocks movement

    ; Neither solid nor breakable - can pass through!
    pop eax
    or al, al             ; Set ZF=0
    ret

ItsBlocking:
    pop eax               ; Clean up stack
    xor al, al            ; Set ZF=1
    ret
IsBlockingTile ENDP
```

Why This Design? Breakable platforms block movement BUT can be destroyed. Solid walls block movement AND cannot be destroyed. By having `IsBlockingTile`, we can check "can I move here?" without worrying about the details!

Game State Management

The State Machine

The game has 6 distinct states:

State	Value	Meaning	Screen
Menu	0	Main menu	MenuScreen
Playing	1	Active gameplay	PlayGame
Paused	2	Game paused	PauseScreen
Level Done	3	Completed level	LevelDoneScreen
Game Over	4	Died, no lives left	GameOverScreen
Victory	5	Beat entire game	WinScreen
Exit	255	Quit to OS	-

Changing state is as simple as:

```
mov gameState, 1      ; Switch to playing
```

On the next frame, the main loop detects this and calls `PlayGame`.

Menu & UI Screens

MenuScreen - The Starting Point

```
MenuScreen PROC
    call Clrscr           ; Clear screen to start fresh

    ; Draw title in yellow
    mov eax, yellow + (black * 16)    ; Foreground + (Background × 16)
    call SetTextColor
    mov dh, 3             ; Row 3
    mov dl, 25            ; Column 25
    call Gotoxy           ; Move cursor to (25, 3)
    mov edx, OFFSET strTitle
    call WriteString      ; Display "SUPER MARIO BROS"

    ; Draw roll number in gray
    mov eax, lightGray + (black * 16)
    call SetTextColor
    mov dh, 5
    mov dl, 48
    call Gotoxy
```

```
    mov edx, OFFSET strRoll
    call WriteString

    ; Draw menu options in white
    mov eax, white + (black * 16)
    call SetTextColor
    ; ... draw each menu item ...

WaitMenu:
    call ReadChar          ; Wait for keypress
                           ; ReadChar is BLOCKING - it waits until a key is pressed
                           ; Returns key in AL

    cmp al, '1'            ; Start new game?
    je StartNew
    cmp al, '2'            ; Continue saved game?
    je ContinueGame
    cmp al, '3'            ; High scores?
    je ShowHigh
    cmp al, '4'            ; Help screen?
    je ShowHelp
    cmp al, '5'            ; Exit?
    je QuitGame
    jmp WaitMenu           ; Invalid key - keep waiting

StartNew:
    call GetName           ; Ask for player name
    ; Reset all saved progress
    mov savedLevel, 1
    mov savedLives, 3
    mov savedScore, 0
    ; ... reset other saved values ...
    call InitGame          ; Initialize a fresh game
    mov gameState, 1       ; Switch to playing state
    ret

ContinueGame:
    call GetName
    call LoadProgress      ; Load saved data from file
    ; Restore values from saved data
    mov al, savedLevel
    mov currentLevel, al
    mov al, savedLives
    mov marioLives, al
    ; ... restore other values ...
    call LoadLevelData     ; Load the saved level
```

```
    mov gameState, 1      ; Start playing
    ret

; ... other menu handlers ...

QuitGame:
    mov gameState, 255    ; Exit flag
    ret
MenuScreen ENDP
```

Color Encoding: Irvine32 uses: `color = foreground + (background × 16)` - Yellow on black = `yellow + (black × 16) = 14 + (0 × 16) = 14` - White on red = `white + (red × 16) = 15 + (4 × 16) = 79`

GetName - Player Input

```
GetName PROC
    call Clrscr
    call ShowCursor       ; Show cursor (blinking line) so user knows they can type

    ; Clear the playerName buffer to avoid garbage
    pushad                ; Save all registers
    mov edi, OFFSET playerName    ; EDI = destination
    mov ecx, 16           ; 16 bytes to clear
    xor al, al            ; AL = 0 (null byte)
    rep stosb             ; Repeat: store AL at [EDI++] until ECX=0
                          ; This is the fastest way to fill memory in assembly!
    popad                 ; Restore registers

    ; Prompt for name
    mov eax, white + (black * 16)
    call SetTextColor
    mov dh, 10
    mov dl, 28
    call Gotoxy
    mov edx, OFFSET strName       ; "Enter your name: "
    call WriteString

    ; Read user input
    mov edx, OFFSET playerName    ; Buffer to store input
    mov ecx, 15           ; Maximum 15 characters (leave 1 for null)
    call ReadString       ; Irvine32 procedure - reads until Enter
                          ; Automatically null-terminates!

    call HideCursor       ; Hide cursor again for gameplay
```

```
    ; Ensure null termination (defensive programming)
    mov BYTE PTR [playerName + 15], 0

    ; Display welcome message
    mov eax, yellow + (black * 16)
    call SetTextColor
    mov dh, 12
    mov dl, 28
    call Gotoxy
    mov edx, OFFSET strHello      ; "Welcome, "
    call WriteString
    mov edx, OFFSET playerName
    call WriteString              ; Display the name
    mov al, '!'
    call WriteChar

    ; Loading animation
    mov eax, lightGreen + (black * 16)
    call SetTextColor
    mov dh, 15
    mov dl, 30
    call Gotoxy
    mov edx, OFFSET strLoad       ; "Loading Level..."
    call WriteString
    mov eax, 1000         ; Wait 1000 milliseconds (1 second)
    call Delay            ; Irvine32 delay procedure
    ret
GetName ENDP
```

REP STOSB Explained: This is one of the most powerful x86 instructions! - **STOSB** = STOre String Byte - stores AL at [EDI] and increments EDI - **REP** = REPeat - repeats the instruction ECX times - **Result:** Fills 16 bytes with zeros in just a few clock cycles!

HelpScreen - Teaching the Player

```
HelpScreen PROC
    call Clrscr

    ; === CONTROLS SECTION ===
    mov eax, yellow + (black * 16)
    call SetTextColor
    mov dh, 2
    mov dl, 50
    call Gotoxy
    mov edx, OFFSET strCtrl       ; "=== CONTROLS ==="
```

```
call WriteString

mov eax, white + (black * 16)
call SetTextColor
; Display each control instruction
mov dh, 4
mov dl, 30
call Gotoxy
mov edx, OFFSET strCtrl1      ; "W / UP Arrow = Jump..."
call WriteString
; ... repeat for other controls ...

; === FEATURES SECTION ===
mov eax, yellow + (black * 16)
call SetTextColor
mov dh, 11
mov dl, 50
call Gotoxy
mov edx, OFFSET strFeat       ; "=== FEATURES ==="
call WriteString

; Colored feature descriptions
mov eax, lightGreen + (black * 16)
call SetTextColor
mov dh, 13
mov dl, 25
call Gotoxy
mov edx, OFFSET strFeat1      ; "o = Coins (100 pts each)..."
call WriteString
; ... more features with different colors ...

; Red warning about score requirement
mov eax, lightRed + (black * 16)
call SetTextColor
mov dh, 23
mov dl, 40
call Gotoxy
mov edx, OFFSET strNeed       ; ">>> Need 700+ score to pass! <<<"
call WriteString

; Wait for any key
mov eax, white + (black * 16)
call SetTextColor
mov dh, 26
mov dl, 50
call Gotoxy
```

```
    mov edx, OFFSET strKey        ; "Press any key..."
    call WriteString
    call ReadChar                 ; Wait for keypress
    ret
HelpScreen ENDP
```

UI Design Philosophy: Notice how we use different colors for different information types: - **Yellow** = Headers/titles (attention-grabbing) - **White** = Normal instructions - **LightGreen** = Positive items (coins, power-ups) - **LightRed** = Warnings/enemies

This creates **visual hierarchy** - players can quickly scan and find what they need!

Game Initialization

InitGame - Fresh Start

This procedure sets everything to its initial state for a brand new game:

```
InitGame PROC
    ; Mario's starting position (bottom-left area)
    mov marioX, 5
    mov marioY, 24
    mov prevMarioX, 5
    mov prevMarioY, 24

    ; Physics state
    mov marioVelY, 0          ; Not moving vertically
    mov marioOnGround, 1      ; Start on ground (not falling)
    mov marioInvinc, 0        ; Not invincible
    mov usedDouble, 0         ; Can double jump
    mov marioDir, 1           ; Facing right
    mov jumpHeld, 0           ; Not holding jump button

    ; Player stats
    mov marioLives, 3         ; 3 lives to start
    mov marioPower, 0         ; Small Mario (no power-ups)
    mov score, 0              ; No score
    mov coins, 0              ; No coins
    mov timer, 150            ; 150 seconds (2.5 minutes)

    ; Level state
    mov currentLevel, 1       ; Start at level 1
    mov goalReached, 0        ; Haven't reached flag yet
```

```
    ; Rendering flags
    mov needFullRedraw, 1     ; Need to draw entire screen
    mov frameCount, 0         ; Frame counter for animations

    ; Special items
    mov freezeActive, 0       ; Freeze power not active
    mov freezeTimer, 0
    mov mushroomActive, 0     ; No mushroom on screen

    ; Clear all entities (enemies, coins, fireballs)
    call ClearEnts
    call ClearFireballs

    ; Load level 1 data
    call LoadLevelData
    ret
InitGame ENDP
```

Why Separate ClearEnts? By putting clearing logic in its own procedure, we can reuse it when loading new levels or respawning Mario without resetting *everything*.

ClearEnts - Entity Cleanup

```
ClearEnts PROC
    pushad                    ; Save all registers

    ; Clear enemy active flags
    xor al, al                ; AL = 0
    mov ecx, MAX_ENEMIES      ; 10 enemies
    mov edi, OFFSET enemyActive     ; Point to array start
    rep stosb                 ; Fill 10 bytes with 0
                              ; This marks all enemies as inactive!

    mov enemyCount, 0         ; No enemies in level

    ; Clear coin active flags
    mov ecx, MAX_COINS        ; 20 coins
    mov edi, OFFSET coinActive
    rep stosb                 ; Fill 20 bytes with 0

    mov coinCount, 0          ; No coins in level

    popad                     ; Restore registers
    ret
ClearEnts ENDP
```

Efficiency Note: We could use a loop with `dec ecx` / `jnz`, but `rep stosb` is **hardware-optimized** and much faster!

LoadLevelData - Level Setup

This is where the level data becomes the living game world:

```
LoadLevelData PROC
    pushad

    ; Clear any existing entities
    call ClearEnts
    call ClearFireballs

    ; Reset Mario to starting position
    mov marioX, 5
    mov marioY, 24
    mov prevMarioX, 5
    mov prevMarioY, 24
    mov marioVelY, 0
    mov marioOnGround, 1
    mov goalReached, 0
    mov needFullRedraw, 1
    mov jumpHeld, 0
    mov usedDouble, 0

    ; Reset special items
    mov freezeActive, 0
    mov freezeTimer, 0
    mov freezeCollected, 0
    mov mushroomActive, 0

    ; Select which level data to load
    mov al, currentLevel      ; AL = 1 or 2
    cmp al, 1
    je UseL1                  ; Load level 1
    jmp UseL2                 ; Load level 2

UseL1:
    mov esi, OFFSET level1    ; ESI = source (level1 array)
    jmp CopyMap

UseL2:
    mov esi, OFFSET level2    ; ESI = source (level2 array)

CopyMap:
```

```
    ; Copy level data into levelMap
    mov edi, OFFSET levelMap  ; EDI = destination
    mov ecx, 3240             ; 120 × 27 = 3240 bytes
    rep movsb                 ; Copy ESI→EDI, repeat ECX times
                              ; MOVSB = MOVe String Byte

    ; Parse the map to find entities
    call FindEntities

    popad
    ret
LoadLevelData ENDP
```

REP MOVSB Magic: This copies 3240 bytes (the entire level!) in one instruction. Behind the scenes, it's roughly:

```
while (ecx > 0) {
    *edi = *esi;
    esi++;
    edi++;
    ecx--;
}
```

But implemented in **hardware** - incredibly fast!

FindEntities - Parsing the Map

This procedure scans through the entire map, finds special characters (coins, enemies, goal), spawns entities, and removes those characters from the map:

```
FindEntities PROC
    pushad

    mov esi, OFFSET levelMap  ; ESI = pointer to map start
    xor ebx, ebx              ; EBX = Y (row counter)

FindRow:
    xor ecx, ecx              ; ECX = X (column counter)

FindCol:
    ; Calculate linear index: Y × 120 + X
    mov eax, ebx              ; EAX = Y
    imul eax, 120             ; EAX = Y × 120
    add eax, ecx              ; EAX = Y×120 + X

    ; Read tile at this position
    mov dl, BYTE PTR [esi + eax]    ; DL = character
```

```
    ; Check what type of entity it is
    cmp dl, 'o'               ; Coin?
    je AddCoin
    cmp dl, 'G'               ; Goomba?
    je AddGoomba
    cmp dl, 'K'               ; Koopa?
    je AddKoopa
    cmp dl, 'F'               ; Flag (goal)?
    je SetGoal
    cmp dl, 'C'               ; Freeze power-up?
    je SetFreeze
    jmp FindNext              ; Not a special character

AddCoin:
    push eax                  ; Save index
    movzx eax, coinCount      ; EAX = current coin count
    cmp al, MAX_COINS         ; At maximum already?
    jge SkipC                 ; Yes - can't add more

    mov edi, eax              ; EDI = index for this coin
    mov coinX[edi], cl        ; Set coin X position (CL = column)
    mov coinY[edi], bl        ; Set coin Y position (BL = row)
    mov coinActive[edi], 1    ; Mark as active (exists)
    inc coinCount             ; Increment total count

SkipC:
    pop eax                   ; Restore index
    mov BYTE PTR [esi + eax], ' '   ; Replace 'o' with space in map
                                    ; (Coin is now an entity, not a tile!)
    jmp FindNext

AddGoomba:
    push eax
    movzx eax, enemyCount
    cmp al, MAX_ENEMIES       ; Room for more enemies?
    jge SkipE

    mov edi, eax              ; EDI = index for this enemy
    mov enemyX[edi], cl       ; Set X position
    mov enemyY[edi], bl       ; Set Y position
    mov enemyPrevX[edi], cl   ; Previous = current initially
    mov enemyPrevY[edi], bl
    mov enemyDir[edi], 0      ; Goombas start moving left
    mov enemyActive[edi], 1   ; Active
    mov enemyType[edi], 0     ; Type 0 = Goomba
    inc enemyCount            ; Increment count
```

```
SkipE:
    pop eax
    mov BYTE PTR [esi + eax], ' '   ; Remove from map
    jmp FindNext

AddKoopa:
    ; Similar to AddGoomba, but with type 1
    push eax
    movzx eax, enemyCount
    cmp al, MAX_ENEMIES
    jge SkipK

    mov edi, eax
    mov enemyX[edi], cl
    mov enemyY[edi], bl
    mov enemyPrevX[edi], cl
    mov enemyPrevY[edi], bl
    mov enemyDir[edi], 1      ; Koopas start moving right
    mov enemyActive[edi], 1
    mov enemyType[edi], 1     ; Type 1 = Koopa
    inc enemyCount

SkipK:
    pop eax
    mov BYTE PTR [esi + eax], ' '
    jmp FindNext

SetGoal:
    mov goalX, cl             ; Remember goal position
    mov goalY, bl
    ; DON'T remove 'F' from map - we want it to display!
    jmp FindNext

SetFreeze:
    mov freezeX, cl           ; Remember freeze power-up position
    mov freezeY, bl
    mov freezeCollected, 0    ; Not collected yet
    ; Keep 'C' in map for display
    jmp FindNext

FindNext:
    inc ecx                   ; Next column
    cmp ecx, 120              ; End of row?
    jl FindCol                ; No - continue scanning this row
```

```
    inc ebx                     ; Next row
    cmp ebx, 27                 ; End of map?
    jl FindRow                  ; No - continue scanning

    ; Done scanning entire map!
    popad
    ret
FindEntities ENDP
```

Why Remove from Map? Coins and enemies are **dynamic** - they move, disappear, etc. Tiles in the map are **static**. By converting them to entities (stored in parallel arrays), we can easily update them each frame without recalculating their positions from the map.

Goal and Freeze Stay in Map: The flag (F) and freeze power-up (C) are rare and static, so it's easier to just check their fixed position and leave them in the map for rendering.

Input Handling

GetInput - Reading Player Commands

This procedure is called every frame and checks if the player pressed any keys:

```
GetInput PROC
    call ReadKey                ; Irvine32 procedure - non-blocking key check
                                ; Returns: AL = ASCII code, ZF = 1 if no key pressed
    jz NoKeyPressed             ; No key? Skip all checks

    ; === JUMP KEYS (W, UP) ===
    cmp al, 'w'
    je JumpKey
    cmp al, 'W'
    je JumpKey
    cmp al, 72                  ; UP arrow (special key code)
    je JumpKey

    ; === MOVE LEFT (A, LEFT) ===
    cmp al, 'a'
    je LeftKey
    cmp al, 'A'
    je LeftKey
    cmp al, 75                  ; LEFT arrow (special key code)
    je LeftKey

    ; === MOVE RIGHT (D, RIGHT) ===
```

```
    cmp al, 'd'
    je RightKey
    cmp al, 'D'
    je RightKey
    cmp al, 77                ; RIGHT arrow (special key code)
    je RightKey

    ; === SHOOT FIREBALL (SPACE) ===
    cmp al, ' '
    je FireKey

    ; === PAUSE (P) ===
    cmp al, 'p'
    je PauseKey
    cmp al, 'P'
    je PauseKey

    ; === QUIT (X) ===
    cmp al, 'x'
    je QuitKey
    cmp al, 'X'
    je QuitKey

    jmp NoKeyPressed          ; Unrecognized key - ignore

; ================================================
JumpKey:
    cmp marioOnGround, 1      ; Is Mario on the ground?
    je DoGroundJump           ; Yes - regular jump

    ; Mario is in the air - check for double jump
    cmp canDoubleJump, 1      ; Is double jump enabled?
    jne NoKeyPressed          ; No - ignore
    cmp usedDouble, 1         ; Already used double jump?
    je NoKeyPressed           ; Yes - can't double jump again

    ; Perform double jump!
    mov marioVelY, DOUBLE_JUMP_VEL    ; -3 (slightly weaker)
    mov usedDouble, 1         ; Mark double jump as used
    mov marioOnGround, 0      ; Still in air
    call PlayJumpSound
    jmp NoKeyPressed

DoGroundJump:
    mov marioVelY, JUMP_VELOCITY      ; -4 (strong jump!)
    mov marioOnGround, 0      ; Now in air
```

```
    mov usedDouble, 0         ; Reset double jump (can use in air now)
    call PlayJumpSound
    jmp NoKeyPressed

; ================================================
LeftKey:
    mov marioDir, 0           ; Face left (direction 0)
    call TryMoveLeft          ; Attempt to move
    jmp NoKeyPressed

RightKey:
    mov marioDir, 1           ; Face right (direction 1)
    call TryMoveRight
    jmp NoKeyPressed

; ================================================
FireKey:
    cmp marioPower, 2         ; Is Mario Fire Mario?
    jne NoKeyPressed          ; No - can't shoot
    call ShootFireball
    jmp NoKeyPressed

; ================================================
PauseKey:
    mov gameState, 2          ; Switch to paused state
    jmp NoKeyPressed

QuitKey:
    mov gameState, 0          ; Back to menu

NoKeyPressed:
    ret
GetInput ENDP
```

ReadKey vs ReadChar: - **ReadKey** is **non-blocking** - returns immediately even if no key pressed (ZF=1) - **ReadChar** is **blocking** - waits until a key is pressed

For gameplay, we need non-blocking so the game doesn't freeze waiting for input!

Arrow Key Codes: - UP = 72 - LEFT = 75 - RIGHT = 77 - DOWN = 80 (not used in this game)

These aren't ASCII - they're **extended key codes** from the BIOS.

Movement Systems

MoveMario - Physics Simulation

This is the heart of the physics engine. It runs every frame (30 times per second) and updates Mario's position based on velocity and gravity:

```
MoveMario PROC
    pushad

    ; ======= GROUND CHECK =======
    ; Even if Mario was on ground last frame, check if he still is!
    ; (Platform might have been destroyed, or Mario walked off edge)

    cmp marioOnGround, 1      ; Were we on ground last frame?
    jne SkipGroundCheck       ; No - already in air, skip check

    ; Check the tile BELOW Mario
    mov bl, marioX            ; BL = Mario's X
    mov bh, marioY            ; BH = Mario's Y
    inc bh                    ; BH = Y+1 (tile below)
    cmp bh, 27                ; Off bottom of screen?
    jge StillOnGround         ; Yes - treat as ground (pit death handled elsewhere)

    call GetTileAt            ; AL = tile below Mario
    call IsBlockingTile       ; Is it solid/breakable?
    jz StillOnGround          ; Yes - still on ground

    ; No ground below! Start falling
    mov marioOnGround, 0
    mov marioVelY, 1          ; Start with downward velocity of 1
    jmp ApplyGravity

StillOnGround:
    jmp NoVerticalMove        ; On ground - don't apply gravity

SkipGroundCheck:

ApplyGravity:
    ; ======= GRAVITY =======
    ; Gravity constantly accelerates Mario downward
    ; velY increases by GRAVITY each frame

    movsx eax, marioVelY      ; EAX = current velocity (SIGNED!)
                              ; MOVSX = MOVe with Sign eXtension
                              ; If marioVelY=-3, EAX becomes -3 (not 253!)
```

```
    add eax, GRAVITY          ; Add gravity (1)
                              ; If velY was -3, now it's -2 (slowing down)
                              ; If velY was 2, now it's 3 (speeding up)

    cmp eax, MAX_FALL_SPEED   ; Reached terminal velocity?
    jle VelOk                 ; No - velocity is ok
    mov eax, MAX_FALL_SPEED   ; Yes - cap it at 4

VelOk:
    mov marioVelY, al         ; Store new velocity

    ; ======= APPLY VERTICAL VELOCITY =======
    movsx eax, marioVelY      ; Get velocity as signed 32-bit
    cmp eax, 0                ; Which direction?
    je NoVerticalMove         ; 0 = not moving
    jl MovingUp               ; Negative = moving up
    call MoveDown             ; Positive = moving down
    jmp NoVerticalMove

MovingUp:
    call MoveUp

NoVerticalMove:

    ; ======= INVINCIBILITY TIMER =======
    cmp invincTimer, 0        ; Any invincibility left?
    je NoInvDec               ; No - skip
    dec invincTimer           ; Decrease timer
    cmp invincTimer, 0        ; Did it reach zero?
    jne NoInvDec              ; No - still invincible
    mov marioInvinc, 0        ; Yes - no longer invincible

NoInvDec:
    popad
    ret
MoveMario ENDP
```

Gravity Formula: Every frame: `velocity = velocity + gravity` - If jumping: velocity starts at -4, becomes -3, -2, -1, 0, 1, 2, 3, 4 (max) - At velocity=0, Mario is at peak of jump (momentarily stationary) - This creates the **parabolic arc** we see in real jumps!

Why MOVSX? Velocity can be negative (jumping up). If we used regular MOV, it would treat -3 as 253 (unsigned). MOVSX properly extends the sign bit, so -3 stays -3.

MoveDown - Falling Physics

```
MoveDown PROC
    pushad

    movsx ecx, marioVelY      ; ECX = velocity (how many pixels to move)
                              ; If velY=3, we move down 3 pixels

MoveDownLoop:
    cmp ecx, 0                ; Moved enough?
    jle MoveDownDone          ; Yes - done

    movzx eax, marioY         ; EAX = current Y
    inc eax                   ; Try moving down 1 pixel
    cmp eax, 26               ; Off bottom of screen?
    jg HitBottom              ; Yes - handle it

    ; Check tile at new position
    mov bl, marioX            ; BL = X
    mov bh, al                ; BH = new Y
    push eax                  ; Save new Y
    call GetTileAt            ; AL = tile at new position
    call IsBlockingTile       ; Would we collide?
    pop eax                   ; Restore new Y
    jz LandOnGround           ; Yes - can't move there, land on it

    ; No collision - move there!
    mov marioY, al            ; Update Mario's Y position
    dec ecx                   ; One pixel moved
    jmp MoveDownLoop          ; Try moving more pixels

LandOnGround:
    ; Mario hit the ground!
    mov marioVelY, 0          ; Stop moving vertically
    mov marioOnGround, 1      ; We're on ground now
    mov usedDouble, 0         ; Reset double jump (can use again)
    jmp MoveDownDone

HitBottom:
    ; Fell off bottom of screen (pit death handled elsewhere)
    mov marioY, 26            ; Clamp to bottom
    mov marioVelY, 0          ; Stop falling

MoveDownDone:
    popad
    ret
```

```
MoveDown ENDP
```

Pixel-by-Pixel Collision: We don't just move Mario by velocity in one step. Instead, we move **1 pixel at a time** and check for collision after each pixel. This prevents Mario from "teleporting through" thin platforms!

Example: If velocity=3, we: 1. Move down 1 pixel, check collision 2. Move down 1 pixel, check collision 3. Move down 1 pixel, check collision

If collision happens at step 2, Mario only moves 2 pixels instead of 3.

MoveUp - Jump Physics with Ceiling Collision

```
MoveUp PROC
    pushad

    movsx ecx, marioVelY      ; ECX = velocity (negative!)
    neg ecx                   ; Make it positive for loop counting
                              ; If velY=-4, ecx becomes 4

MoveUpLoop:
    cmp ecx, 0                ; Moved enough?
    jle MoveUpDone            ; Yes - done

    movzx eax, marioY         ; EAX = current Y
    dec eax                   ; Try moving up 1 pixel
    cmp eax, 1                ; Hit ceiling (top of screen)?
    jl HitCeiling             ; Yes - can't go higher

    ; Check tile at new position
    mov bl, marioX            ; BL = X
    mov bh, al                ; BH = new Y
    push eax                  ; Save new Y
    push ecx                  ; Save loop counter
    call GetTileAt            ; AL = tile at new position

    ; === SPECIAL TILES ===
    cmp al, '?'               ; Mystery block?
    je HitQuestionBlock       ; Yes - trigger power-up!

    cmp al, '='               ; Breakable platform?
    je HitBreakable           ; Yes - destroy it!

    ; Regular collision check
    call IsSolidTile          ; Is it solid?
    pop ecx                   ; Restore loop counter
    pop eax                   ; Restore new Y
```

```
    jz HitCeiling             ; Solid - bonk head on ceiling

    ; No collision - move there!
    mov marioY, al
    dec ecx                   ; One pixel moved
    jmp MoveUpLoop

; ============================================
HitQuestionBlock:
    pop ecx                   ; Clean up stack
    pop eax
    push eax                  ; Need position for tile change
    push ebx

    ; Change '?' to '-' in the map (used block)
    movzx eax, bh             ; EAX = Y
    imul eax, MAP_W           ; EAX = Y × 120
    movzx edx, bl             ; EDX = X
    add eax, edx              ; EAX = index
    mov esi, OFFSET levelMap
    mov BYTE PTR [esi + eax], '-'    ; Replace ? with -

    add score, 50             ; Bonus points for hitting block

    ; Redraw the block on screen
    push ebx
    mov eax, gray + (black * 16)
    call SetTextColor
    mov dl, bl                ; Column
    mov dh, bh                ; Row
    inc dh                    ; (Adjust for HUD offset)
    call Gotoxy
    mov al, '-'
    call WriteChar
    pop ebx

    ; === SPAWN MUSHROOM ===
    cmp marioPower, 0         ; Is Mario small?
    jne NoPowerGive           ; No - give different power-up

    ; Spawn mushroom above block
    mov al, bl                ; X position
    mov mushroomX, al
    mov mushroomPrevX, al
    mov al, bh                ; Y position
    dec al                    ; Above the block
```

```
    mov mushroomY, al
    mov mushroomPrevY, al
    mov mushroomActive, 1     ; Mushroom exists now
    mov mushroomDir, 1        ; Moving right
    jmp PowerGiven

NoPowerGive:
    ; Give fire power if already big
    cmp marioPower, 1         ; Is Mario big?
    jne PowerGiven            ; No - already fire
    mov marioPower, 2         ; Upgrade to Fire Mario!

PowerGiven:
    call PlayPowerupSound
    pop ebx
    pop eax
    jmp HitCeiling            ; Stop upward movement

; ================================================
HitBreakable:
    pop ecx
    pop eax
    push eax
    push ebx

    ; Remove '=' from the map
    movzx eax, bh
    imul eax, MAP_W
    movzx edx, bl
    add eax, edx
    mov esi, OFFSET levelMap
    mov BYTE PTR [esi + eax], ' '    ; Replace with space

    add score, 25             ; Bonus for breaking platform

    ; Erase from screen
    mov eax, black + (black * 16)
    call SetTextColor
    mov dl, bl
    mov dh, bh
    inc dh
    call Gotoxy
    mov al, ' '
    call WriteChar

    call PlayStompSound       ; Satisfying crunch sound
```

```
    pop ebx
    pop eax

HitCeiling:
    ; Bonked head - stop jumping immediately
    mov marioVelY, 1          ; Start falling (downward velocity)
    mov marioOnGround, 0      ; In air now

MoveUpDone:
    popad
    ret
MoveUp ENDP
```

Mystery Block Logic: When Mario hits a `?` from below: 1. Change map tile to `-` (used block) 2. Award points 3. Check Mario's power level 4. Spawn appropriate power-up (mushroom for small, fire for big)

Breakable Platform: Unlike mystery blocks, these **disappear entirely** when hit from below, creating new paths!

TryMoveLeft / TryMoveRight - Horizontal Movement

```
TryMoveLeft PROC
    pushad

    movzx eax, marioX         ; EAX = current X
    cmp eax, 2                ; At left edge?
    jle CantMoveLeft          ; Yes - can't move further left
                              ; (Edge is at X=2 because X=0-1 are border walls)

    ; Check tile to the left
    mov bl, marioX
    dec bl                    ; BL = X - 1
    mov bh, marioY            ; BH = Y
    call GetTileAt            ; AL = tile to the left
    call IsBlockingTile       ; Is it solid?
    jz CantMoveLeft           ; Yes - can't move there

    ; Clear to move left!
    dec marioX                ; X = X - 1

CantMoveLeft:
    popad
    ret
TryMoveLeft ENDP

TryMoveRight PROC
```

```
    pushad

    movzx eax, marioX
    cmp eax, 117              ; At right edge?
    jge CantMoveRight         ; Yes - can't move further
                              ; (117 because screen is 120 wide, 118-119 are border)

    ; Check tile to the right
    mov bl, marioX
    inc bl                    ; BL = X + 1
    mov bh, marioY
    call GetTileAt
    call IsBlockingTile
    jz CantMoveRight

    ; Clear to move right!
    inc marioX

CantMoveRight:
    popad
    ret
TryMoveRight ENDP
```

Instantaneous Movement: Unlike vertical movement (which uses velocity), horizontal movement is **immediate** - press D and Mario moves 1 pixel right instantly. This makes platformers feel more responsive!

MoveEnemies - AI Patrol

Enemies have simple but effective AI: walk in one direction until hitting a wall, then turn around.

```
MoveEnemies PROC
    pushad

    movzx ecx, enemyCount     ; ECX = total enemies in level
    cmp ecx, 0                ; Any enemies?
    je NoEnMove               ; No - nothing to do

    xor ebx, ebx              ; EBX = current enemy index (0)

EnMoveLoop:
    cmp enemyActive[ebx], 0   ; Is this enemy active?
    je NextEnemy              ; No - skip it

    ; Save previous position (for rendering)
    mov al, enemyX[ebx]
```

```
    mov enemyPrevX[ebx], al
    mov al, enemyY[ebx]
    mov enemyPrevY[ebx], al

    ; Check direction
    mov al, enemyDir[ebx]
    cmp al, 0                   ; Moving left?
    je EnMoveLeft               ; Yes

    ; === MOVE RIGHT ===
    movzx edx, enemyX[ebx]      ; EDX = current X
    inc edx                     ; EDX = X + 1
    cmp edx, 117                ; Hit right edge?
    jge EnReverseToLeft         ; Yes - turn around

    ; Check tile to the right
    push ebx                    ; Save index
    push edx
    mov al, enemyY[ebx]
    mov bh, al                  ; BH = enemy's Y
    mov bl, dl                  ; BL = new X
    call GetTileAt              ; AL = tile at new position
    call IsBlockingTile         ; Is it solid?
    pop edx
    pop ebx
    jz EnReverseToLeft          ; Yes - turn around

    ; Clear to move right
    inc enemyX[ebx]             ; X = X + 1
    jmp NextEnemy

EnReverseToLeft:
    mov enemyDir[ebx], 0        ; Change direction to left
    jmp NextEnemy

    ; === MOVE LEFT ===
EnMoveLeft:
    movzx edx, enemyX[ebx]
    dec edx                     ; EDX = X - 1
    cmp edx, 2                  ; Hit left edge?
    jle EnReverseToRight        ; Yes - turn around

    ; Check tile to the left
    push ebx
    push edx
    mov al, enemyY[ebx]
```

```
    mov bh, al
    mov bl, dl
    call GetTileAt
    call IsBlockingTile
    pop edx
    pop ebx
    jz EnReverseToRight       ; Wall - turn around

    ; Clear to move left
    dec enemyX[ebx]
    jmp NextEnemy

EnReverseToRight:
    mov enemyDir[ebx], 1      ; Change direction to right

NextEnemy:
    inc ebx                   ; Next enemy
    cmp ebx, ecx              ; Processed all enemies?
    jl EnMoveLoop             ; No - continue

NoEnMove:
    popad
    ret
MoveEnemies ENDP
```

Frame-Rate Throttling: In `PlayGame`, enemies only move every other frame:

```
mov al, frameCount
and al, 1                 ; AL = frameCount % 2
cmp al, 0                 ; Even frame?
jne SkipEnMove            ; No - skip enemy movement
call MoveEnemies          ; Yes - move enemies
```

This makes enemies move at **half speed** of Mario, making the game easier!

MoveMushroom - Power-up Movement

The mushroom power-up moves similarly to enemies (back and forth), but horizontally:

```
MoveMushroom PROC
    pushad

    cmp mushroomActive, 0     ; Does mushroom exist?
    je NoMushMove             ; No - nothing to do

    ; Check direction
    cmp mushroomDir, 0
```

```
    je MushLeft               ; 0 = moving left

    ; === MOVE RIGHT ===
    movzx eax, mushroomX
    inc eax                   ; Try moving right
    cmp eax, 117              ; Hit edge?
    jge MushReverseL          ; Yes - reverse

    ; Check collision
    mov bl, al                ; BL = new X
    mov bh, mushroomY         ; BH = Y
    push eax
    call GetTileAt
    call IsBlockingTile
    pop eax
    jz MushReverseL           ; Hit wall - reverse

    ; Move right
    mov mushroomX, al
    jmp NoMushMove

MushReverseL:
    mov mushroomDir, 0        ; Reverse to left
    jmp NoMushMove

    ; === MOVE LEFT ===
MushLeft:
    movzx eax, mushroomX
    dec eax                   ; Try moving left
    cmp eax, 2                ; Hit edge?
    jle MushReverseR          ; Yes - reverse

    ; Check collision
    mov bl, al
    mov bh, mushroomY
    push eax
    call GetTileAt
    call IsBlockingTile
    pop eax
    jz MushReverseR           ; Hit wall - reverse

    ; Move left
    mov mushroomX, al
    jmp NoMushMove

MushReverseR:
```

```
    mov mushroomDir, 1        ; Reverse to right

NoMushMove:
    popad
    ret
MoveMushroom ENDP
```

Mushrooms vs Enemies: The code is nearly identical! In a real game engine, we'd use **inheritance** or **components** to avoid duplication. In assembly, we just copy-paste and modify slightly.

Collision Detection

CheckCoins - Coin Collection

```
CheckCoins PROC
    pushad

    movzx ecx, coinCount      ; ECX = total coins in level
    cmp ecx, 0                ; Any coins?
    je NoCoinHit              ; No - nothing to check

    xor ebx, ebx              ; EBX = current coin index

CoinLoop:
    cmp coinActive[ebx], 0    ; Is this coin active?
    je NextCoin               ; No - skip it

    ; === CHECK X COLLISION ===
    movzx eax, marioX         ; EAX = Mario's X
    movzx edx, coinX[ebx]     ; EDX = coin's X
    sub eax, edx              ; EAX = difference

    ; Coin is collected if within 1 tile horizontally
    cmp eax, 1                ; Is Mario to the right by 1 tile?
    jg NextCoin               ; No - too far right
    cmp eax, -1               ; Is Mario to the left by 1 tile?
    jl NextCoin               ; No - too far left

    ; === CHECK Y COLLISION ===
    movzx eax, marioY         ; EAX = Mario's Y
    movzx edx, coinY[ebx]     ; EDX = coin's Y
    sub eax, edx              ; EAX = difference

    cmp eax, 1                ; Within 1 tile vertically?
```

```
    jg NextCoin               ; No - too far down
    cmp eax, -1               ; Within 1 tile vertically?
    jl NextCoin               ; No - too far up

    ; === COLLECT COIN ===
    mov coinActive[ebx], 0    ; Deactivate coin
    inc coins                 ; Increment coin counter
    add score, 100            ; 100 points per coin
    call PlayCoinSound        ; Satisfying "ding!"

    ; Erase coin from screen immediately
    push ebx
    mov eax, black + (black * 16)
    call SetTextColor
    mov dl, coinX[ebx]
    mov dh, coinY[ebx]
    inc dh                    ; Adjust for HUD row
    call Gotoxy
    mov al, ' '
    call WriteChar
    pop ebx

    ; === EXTRA LIFE CHECK ===
    cmp coins, 50             ; Collected 50 coins?
    jl NextCoin               ; No - continue

    mov coins, 0              ; Reset coin counter
    inc marioLives            ; Extra life!
    ; (Could play special sound here)

NextCoin:
    inc ebx                   ; Next coin
    cmp ebx, ecx              ; Checked all coins?
    jl CoinLoop               ; No - continue

NoCoinHit:
    popad
    ret
CheckCoins ENDP
```

Collision Tolerance: We check if distance is within **1 tile** in both X and Y. This gives a generous collision box - makes the game feel fair. If we required **exact** position, it would be frustrating!

50-Coin Bonus: This is a classic Mario mechanic - collect 50 coins for an extra life, then counter resets. Encourages exploration!

CheckEnemies - The Most Complex Collision

This procedure handles: 1. **Side collision** = Mario takes damage 2. **Top collision** = Mario stomps enemy

```
CheckEnemies PROC
    pushad

    cmp marioInvinc, 1        ; Is Mario invincible?
    je NoEnHit                ; Yes - skip all collision checks

    movzx ecx, enemyCount
    cmp ecx, 0
    je NoEnHit

    xor ebx, ebx              ; EBX = enemy index

EnHitLoop:
    cmp enemyActive[ebx], 0
    je NextEn

    ; === CHECK X COLLISION ===
    movzx eax, marioX
    movzx edx, enemyX[ebx]
    sub eax, edx              ; EAX = X difference

    cmp eax, 1                ; Within collision range?
    jg NextEn                 ; No - too far right
    cmp eax, -1               ; Within collision range?
    jl NextEn                 ; No - too far left

    ; X collision detected! Now check Y to determine type

    ; === CHECK Y COLLISION ===
    movzx eax, marioY         ; EAX = Mario's Y
    movzx edx, enemyY[ebx]    ; EDX = enemy's Y
    sub eax, edx              ; EAX = Y difference

    ; Analyze the Y difference:
    ; If Mario is 1-2 tiles ABOVE enemy and falling → STOMP
    ; If Mario is at same level or below → DAMAGE

    cmp eax, -2               ; Is Mario 2+ tiles above?
    jl NextEn                 ; Yes - too high, no collision

    cmp eax, -1               ; Is Mario 1-2 tiles above?
```

```
    jle StompEnemy            ; Yes - STOMP!

    ; Mario is at same level or below - DAMAGE
    cmp eax, 0
    je DamageFromSide
    cmp eax, 1
    je DamageFromSide
    jmp NextEn                ; Shouldn't happen, but safety check

; ================================================
DamageFromSide:
    call TakeDmg              ; Mario gets hurt
    jmp NoEnHit               ; Stop checking (invincibility frames)

; ================================================
StompEnemy:
    ; Check enemy type
    cmp enemyType[ebx], 1     ; Is it a Koopa (type 1)?
    je StompKoopa             ; Yes - special handling

    ; It's a Goomba (type 0) - just defeat it
    mov enemyActive[ebx], 0   ; Remove enemy
    add score, 100            ; 100 points
    mov marioVelY, -2         ; Bounce Mario upward (satisfying!)
    call PlayStompSound

    ; Erase enemy from screen
    push ebx
    mov eax, black + (black * 16)
    call SetTextColor
    mov dl, enemyX[ebx]
    mov dh, enemyY[ebx]
    inc dh
    call Gotoxy
    mov al, ' '
    call WriteChar
    pop ebx
    jmp NextEn

StompKoopa:
    ; Convert Koopa to Shell (type 2)
    mov enemyType[ebx], 2     ; Type 2 = Shell
    add score, 100            ; 100 points
    mov marioVelY, -2         ; Bounce Mario
    call PlayStompSound
    ; Shell stays active but now stationary (could add kick mechanic!)
```

```
    jmp NextEn

NextEn:
    inc ebx
    cmp ebx, ecx
    jl EnHitLoop

NoEnHit:
    popad
    ret
CheckEnemies ENDP
```

Stomp Detection Math:

```
If Mario.Y = 10 and Enemy.Y = 12:
    difference = 10 - 12 = -2 (Mario is 2 above)
    -2 <= -1 → TRUE → STOMP!

If Mario.Y = 12 and Enemy.Y = 12:
    difference = 12 - 12 = 0 (same level)
    0 == 0 → TRUE → DAMAGE!
```

Goomba vs Koopa: - **Goomba**: Defeated immediately when stomped (removed from game) - **Koopa**: Converts to stationary shell (type 2) when stomped

This difference adds **variety** to enemy behaviors!

TakeDmg - Damage System

```
TakeDmg PROC
    cmp marioInvinc, 1        ; Already invincible?
    je NoDmg                  ; Yes - no damage

    cmp marioPower, 0         ; Is Mario small?
    je MustDie                ; Yes - death

    ; Mario has power-up - lose it instead of dying
    dec marioPower            ; Fire→Big or Big→Small
    mov marioInvinc, 1        ; Activate invincibility
    mov invincTimer, 60       ; 60 frames = 2 seconds
    call PlayDamageSound
    jmp NoDmg

MustDie:
    call DieMario             ; Small Mario died

NoDmg:
```

```
    ret
TakeDmg ENDP
```

Invincibility Frames: After taking damage, Mario is invincible for 2 seconds. This prevents **instant death** from continuous collision (enemy touching Mario multiple times).

The timer counts down in `MoveMario`:

```
cmp invincTimer, 0
je NoInvDec
dec invincTimer
cmp invincTimer, 0
jne NoInvDec
mov marioInvinc, 0        ; Invincibility expired
```

DieMario - Death Handler

```
DieMario PROC
    dec marioLives            ; Lose a life
    call PlayDamageSound

    cmp marioLives, 0         ; Any lives left?
    jle GameEnd               ; No - game over

    ; Respawn
    mov needFullRedraw, 1     ; Redraw entire screen
    mov timer, 150            ; Reset timer to 150 seconds
    call LoadLevelData        ; Reload level (fresh enemies, coins, etc.)
    ret

GameEnd:
    mov gameState, 4          ; Switch to Game Over state
    ret
DieMario ENDP
```

Respawn Behavior: When Mario dies but has lives remaining: 1. Timer resets to 150 (full time) 2. Level is completely reloaded (enemies respawn, coins reappear) 3. Score, power-ups, and coins are **preserved**

This is **forgiving** but not too easy - you keep your progress but enemies reset.

Rendering Engine

RenderFrame - The Rendering Pipeline

This is called 30 times per second and decides what to draw:

```
RenderFrame PROC
    cmp needFullRedraw, 1     ; Do we need full screen refresh?
    je DoFullRedraw           ; Yes - redraw everything

    ; === INCREMENTAL RENDERING ===
    ; Only redraw what changed
    call DrawHUD              ; Update HUD (score, lives, etc.)
    call ErasePrevMario       ; Erase Mario's old position
    call DrawEnemySprites     ; Draw enemies at new positions
    call DrawCoinsSprites     ; Draw coins (static, but cheap)
    call DrawFireballSprites  ; Draw fireballs
    call DrawMushroomSprite   ; Draw mushroom if exists
    call DrawFreezeSprite     ; Draw freeze power if exists
    call DrawMarioSprite      ; Draw Mario at new position
    call SavePrevPositions    ; Remember positions for next frame
    jmp RenderDone

DoFullRedraw:
    ; === FULL SCREEN REDRAW ===
    ; Clear screen and redraw EVERYTHING
    call Clrscr               ; Clear entire screen
    call DrawHUD
    call DrawMap              ; Draw the entire level tilemap
    call DrawCoinsSprites
    call DrawEnemySprites
    call DrawFireballSprites
    call DrawMushroomSprite
    call DrawFreezeSprite
    call DrawMarioSprite
    call SavePrevPositions
    mov needFullRedraw, 0     ; Clear flag

RenderDone:
    ret
RenderFrame ENDP
```

Optimization Strategy: Full screen redraws are **expensive** (3000+ characters). We only do full redraws when: - Level loads - Unpausing - After showing a message

Most frames use **incremental rendering** - only erase/redraw entities that moved. This is much faster!

DrawHUD - Status Display

```
DrawHUD PROC
    pushad
```

```
mov dh, 0                  ; Row 0 (top of screen)
mov dl, 0                  ; Column 0 (left edge)
call Gotoxy

; === PLAYER NAME ===
mov eax, lightCyan + (black * 16)
call SetTextColor
mov edx, OFFSET strPlayer ; "Player: "
call WriteString
mov edx, OFFSET playerName
call WriteString

; === SCORE ===
mov eax, white + (black * 16)
call SetTextColor
mov al, ' '                ; Space separator
call WriteChar
call WriteChar
mov al, 'S'
call WriteChar
mov al, 'C'
call WriteChar
mov al, 'O'
call WriteChar
mov al, 'R'
call WriteChar
mov al, 'E'
call WriteChar
mov al, ':'
call WriteChar
mov eax, score             ; Load score into EAX
call WriteDec              ; Write decimal number

; === COINS ===
mov eax, yellow + (black * 16)
call SetTextColor
mov al, ' '
call WriteChar
call WriteChar
mov al, 'o'                ; Coin symbol
call WriteChar
mov al, ':'
call WriteChar
movzx eax, coins           ; Load coin count
call WriteDec
```

```
; === LEVEL ===
mov eax, white + (black * 16)
call SetTextColor
mov al, ' '
call WriteChar
call WriteChar
mov al, 'L'
call WriteChar
mov al, 'V'
call WriteChar
mov al, 'L'
call WriteChar
mov al, ':'
call WriteChar
movzx eax, currentLevel
call WriteDec
mov al, '/'
call WriteChar
mov al, '2'
call WriteChar            ; "LVL:1/2" or "LVL:2/2"

; === TIMER ===
mov al, ' '
call WriteChar
call WriteChar
mov al, 'T'
call WriteChar
mov al, 'I'
call WriteChar
mov al, 'M'
call WriteChar
mov al, 'E'
call WriteChar
mov al, ':'
call WriteChar
movzx eax, timer          ; Load timer
call WriteDec

; === LIVES ===
mov eax, red + (black * 16)
call SetTextColor
mov al, ' '
call WriteChar
call WriteChar
mov al, 'M'
```

```
    call WriteChar
    mov al, 'x'
    call WriteChar            ; "Mx3" = Mario × 3 lives
    movzx eax, marioLives
    call WriteDec

    ; === POWER INDICATOR ===
    mov al, ' '
    call WriteChar

    cmp marioPower, 2         ; Fire Mario?
    je ShowFire
    cmp marioPower, 1         ; Big Mario?
    je ShowBig
    jmp ShowSmall             ; Small Mario

ShowFire:
    mov eax, lightRed + (black * 16)
    call SetTextColor
    mov al, 'F'
    call WriteChar
    mov al, 'I'
    call WriteChar
    mov al, 'R'
    call WriteChar
    mov al, 'E'
    call WriteChar
    jmp AfterPower

ShowBig:
    mov eax, lightGreen + (black * 16)
    call SetTextColor
    mov al, 'B'
    call WriteChar
    mov al, 'I'
    call WriteChar
    mov al, 'G'
    call WriteChar
    mov al, ' '
    call WriteChar
    jmp AfterPower

ShowSmall:
    ; No text for small Mario (4 spaces for alignment)
    mov al, ' '
    call WriteChar
```

```
    call WriteChar
    call WriteChar
    call WriteChar

AfterPower:
    ; === FREEZE INDICATOR ===
    cmp freezeActive, 1       ; Is freeze power active?
    jne NoFreezeInd           ; No - skip

    mov eax, lightCyan + (blue * 16)  ; Cyan on blue (icy!)
    call SetTextColor
    mov al, ' '
    call WriteChar
    mov al, 'F'
    call WriteChar
    mov al, 'R'
    call WriteChar
    mov al, 'E'
    call WriteChar
    mov al, 'E'
    call WriteChar
    mov al, 'Z'
    call WriteChar
    mov al, 'E'
    call WriteChar
    mov al, '!'
    call WriteChar            ; "FREEZE!"

NoFreezeInd:
    ; Clear any remaining space (in case freeze just deactivated)
    mov eax, white + (black * 16)
    call SetTextColor
    mov al, ' '
    mov ecx, 10               ; 10 spaces to clear rest of line
ClearRest:
    call WriteChar
    loop ClearRest

    popad
    ret
DrawHUD ENDP
```

Layout Design: The HUD is carefully laid out for **information hierarchy**: 1. **Left side**: Player identity and score (most important) 2. **Middle**: Level/time (context) 3. **Right side**: Lives and power (survival info)

Dynamic Updates: The HUD is redrawn every frame, so it always shows

current info. This is fine because it's only 1 row of 120 characters.

DrawMap - Rendering the Level

```
DrawMap PROC
    pushad

    mov esi, OFFSET levelMap  ; ESI = pointer to map data
    xor ebx, ebx              ; EBX = row counter (Y)

MapRow:
    mov dh, bl                ; DH = row
    inc dh                    ; +1 because row 0 is HUD
    xor dl, dl                ; DL = 0 (start at left column)
    call Gotoxy               ; Position cursor

    xor ecx, ecx              ; ECX = column counter (X)

MapCol:
    ; Calculate index: Y × 120 + X
    mov eax, ebx              ; EAX = Y
    imul eax, 120             ; EAX = Y × 120
    add eax, ecx              ; EAX = Y×120 + X

    ; Read character at this position
    mov al, BYTE PTR [esi + eax]

    ; Select color based on tile type
    cmp al, '#'               ; Brick wall?
    je ColBrick
    cmp al, '?'               ; Mystery block?
    je ColQ
    cmp al, '='               ; Breakable platform?
    je ColPlat
    cmp al, '-'               ; Used mystery block?
    je ColUsed
    cmp al, '~'               ; Lava? (not used in current levels)
    je ColLava
    cmp al, 'F'               ; Flag (goal)?
    je ColFlag
    cmp al, '['               ; Castle wall left?
    je ColCastle
    cmp al, ']'               ; Castle wall right?
    je ColCastle
    cmp al, 'C'               ; Freeze power-up?
    je ColFreeze
```

```
    jmp ColNorm               ; Space or unknown - white

ColBrick:
    push eax
    mov eax, red + (black * 16)       ; Red bricks
    call SetTextColor
    pop eax
    jmp DoChar

ColQ:
    push eax
    mov eax, yellow + (black * 16)    ; Yellow mystery blocks
    call SetTextColor
    pop eax
    jmp DoChar

ColPlat:
    push eax
    mov eax, brown + (black * 16)     ; Brown breakable platforms
    call SetTextColor
    pop eax
    jmp DoChar

ColUsed:
    push eax
    mov eax, gray + (black * 16)      ; Gray used blocks
    call SetTextColor
    pop eax
    jmp DoChar

ColLava:
    push eax
    mov eax, lightRed + (red * 16)    ; Bright red lava
    call SetTextColor
    pop eax
    jmp DoChar

ColFlag:
    push eax
    mov eax, white + (green * 16)     ; White flag on green
    call SetTextColor
    pop eax
    jmp DoChar

ColCastle:
    push eax
```

```
    mov eax, lightGray + (black * 16) ; Gray castle
    call SetTextColor
    pop eax
    jmp DoChar

ColFreeze:
    push eax
    mov eax, lightCyan + (blue * 16)  ; Cyan freeze power
    call SetTextColor
    pop eax
    jmp DoChar

ColNorm:
    push eax
    mov eax, white + (black * 16)     ; White (empty space)
    call SetTextColor
    pop eax

DoChar:
    call WriteChar            ; Output the character

    inc ecx                   ; Next column
    cmp ecx, 120              ; End of row?
    jl MapCol                 ; No - continue this row

    inc ebx                   ; Next row
    cmp ebx, 27               ; End of map?
    jl MapRow                 ; No - continue

    popad
    ret
DrawMap ENDP
```

Color Coding: Each tile type has a distinct color. This isn't just aesthetic - it provides **visual information**: - Red = danger (walls, can't pass) - Yellow = interactive (mystery blocks) - Brown = destructible (breakable platforms) - Green = goal (flag)

DrawMarioSprite - The Hero

```
DrawMarioSprite PROC
    pushad

    mov dl, marioX            ; DL = column
    mov dh, marioY            ; DH = row
    inc dh                    ; +1 for HUD offset
```

```
    call Gotoxy               ; Position cursor at Mario's location

    ; Select Mario's appearance based on state
    cmp marioInvinc, 1        ; Is Mario invincible?
    je InvincibleMario        ; Yes - special flashing colors

    cmp marioPower, 2         ; Is Mario Fire Mario?
    je FireMario
    cmp marioPower, 1         ; Is Mario Big Mario?
    je BigMario

    ; Small Mario (default)
    mov eax, cyan + (black * 16)      ; Cyan 'm'
    call SetTextColor
    mov al, 'm'               ; Lowercase m (small!)
    call WriteChar
    jmp MarioDone

BigMario:
    mov eax, lightCyan + (black * 16) ; Light cyan 'M'
    call SetTextColor
    mov al, 'M'               ; Uppercase M (big!)
    call WriteChar
    jmp MarioDone

FireMario:
    mov eax, white + (red * 16)       ; White M on red (fire!)
    call SetTextColor
    mov al, 'M'
    call WriteChar
    jmp MarioDone

InvincibleMario:
    mov eax, yellow + (magenta * 16)  ; Yellow on magenta (flashing!)
    call SetTextColor
    mov al, 'M'
    call WriteChar

MarioDone:
    popad
    ret
DrawMarioSprite ENDP
```

Visual Progression: 1. `m` (cyan) = Small Mario 2. `M` (light cyan) = Big Mario 3. `M` (white on red) = Fire Mario 4. `M` (yellow on magenta) = Invincible (any size)

The size difference (lowercase vs uppercase) and color changes give instant visual feedback about Mario's power level!

DrawEnemySprites - Enemy Rendering

```
DrawEnemySprites PROC
    pushad

    movzx ecx, enemyCount     ; ECX = total enemies
    cmp ecx, 0
    je NoEnDraw

    xor ebx, ebx              ; EBX = enemy index

EnDraw:
    cmp enemyActive[ebx], 0   ; Is this enemy active?
    je SkipEnDraw             ; No - skip

    ; === ERASE OLD POSITION (if moved) ===
    mov al, enemyPrevX[ebx]
    cmp al, enemyX[ebx]       ; Did X change?
    je NoEraseEn              ; No - don't erase

    push ebx
    mov eax, white + (black * 16)     ; White (effectively erases)
    call SetTextColor
    mov dl, enemyPrevX[ebx]
    mov dh, enemyPrevY[ebx]
    inc dh
    call Gotoxy
    mov al, ' '
    call WriteChar
    pop ebx

NoEraseEn:
    ; === DRAW AT NEW POSITION ===
    mov dl, enemyX[ebx]
    mov dh, enemyY[ebx]
    inc dh
    call Gotoxy

    ; Check enemy type
    cmp enemyType[ebx], 2     ; Shell?
    je DrawShell
    cmp enemyType[ebx], 1     ; Koopa?
    je DrawK
```

```
    ; Goomba (type 0)
    mov eax, brown + (black * 16)     ; Brown Goomba
    call SetTextColor
    mov al, 'G'
    call WriteChar
    jmp SkipEnDraw

DrawK:
    mov eax, green + (black * 16)     ; Green Koopa
    call SetTextColor
    mov al, 'K'
    call WriteChar
    jmp SkipEnDraw

DrawShell:
    mov eax, green + (black * 16)     ; Green Shell
    call SetTextColor
    mov al, 'S'
    call WriteChar

SkipEnDraw:
    inc ebx                   ; Next enemy
    cmp ebx, ecx
    jl EnDraw

NoEnDraw:
    popad
    ret
DrawEnemySprites ENDP
```

Erase-Then-Draw: For enemies that moved, we: 1. Draw a space at old position (erases old sprite) 2. Draw enemy character at new position

If the enemy didn't move, we skip erasing (optimization).

Enemy Types: - `G` (brown) = Goomba (basic enemy) - `K` (green) = Koopa (becomes shell when stomped) - `S` (green) = Shell (stomped Koopa)

File I/O System

SaveScore - Persistent High Scores

The game saves the **top 3 scores** and player names to a file:

```
SaveScore PROC
    pushad
```

```
    ; First, update the top scores array with current score
    call UpdateTopScores

    ; Create/open file for writing
    mov edx, OFFSET fileScore ; "mario_hs.dat"
    call CreateOutputFile     ; Irvine32 procedure
    cmp eax, INVALID_HANDLE_VALUE
    je NoSave                 ; Failed to create - skip

    mov fileHandle, eax       ; Save file handle

    ; === WRITE 3 SCORES (12 bytes total) ===
    mov edx, OFFSET topScores ; Point to array start
    mov ecx, 12               ; 3 DWORDs = 12 bytes
    call WriteToFile          ; Irvine32 procedure

    ; === WRITE 3 NAMES (48 bytes total) ===
    mov edx, OFFSET topNames  ; Point to names start
    mov ecx, 48               ; 3 names × 16 bytes each
    call WriteToFile

    ; Close file
    mov eax, fileHandle
    call CloseFile

NoSave:
    popad
    ret
SaveScore ENDP
```

File Format:

```
Bytes 0-3:    Score 1 (DWORD)
Bytes 4-7:    Score 2 (DWORD)
Bytes 8-11:   Score 3 (DWORD)
Bytes 12-27:  Name 1 (16 chars)
Bytes 28-43:  Name 2 (16 chars)
Bytes 44-59:  Name 3 (16 chars)
Total: 60 bytes
```

UpdateTopScores - Ranking System

This procedure determines if the current score made the top 3, and if so, shifts the rankings:

```
UpdateTopScores PROC
    pushad

    mov eax, score            ; EAX = current score

    ; === CHECK 1ST PLACE ===
    cmp eax, topScores[0]     ; Is current score > 1st place?
    jle Check2nd              ; No - check 2nd place

    ; New 1st place! Shift everyone down:
    ; 2nd → 3rd
    ; 1st → 2nd
    ; Current → 1st

    ; Shift 2nd place score to 3rd
    mov ebx, topScores[4]     ; EBX = 2nd place score
    mov topScores[8], ebx     ; Store in 3rd place

    ; Copy 2nd place name to 3rd place
    mov ecx, 0
CopyName1to2:
    cmp ecx, 16
    jge DoneCopy1to2
    mov bl, BYTE PTR [topNames + ecx + 16]  ; Read from slot 1
    mov BYTE PTR [topNames + ecx + 32], bl  ; Write to slot 2
    inc ecx
    jmp CopyName1to2
DoneCopy1to2:
    mov BYTE PTR [topNames + 32 + 15], 0    ; Null terminate

    ; Shift 1st place score to 2nd
    mov ebx, topScores[0]
    mov topScores[4], ebx

    ; Copy 1st place name to 2nd place
    mov ecx, 0
CopyName0to1:
    cmp ecx, 16
    jge DoneCopy0to1
    mov bl, BYTE PTR [topNames + ecx]       ; Read from slot 0
    mov BYTE PTR [topNames + ecx + 16], bl  ; Write to slot 1
    inc ecx
    jmp CopyName0to1
DoneCopy0to1:
    mov BYTE PTR [topNames + 16 + 15], 0
```

```
    ; Set new 1st place
    mov topScores[0], eax     ; Store current score
    mov highScore, eax        ; Update highScore variable

    ; Copy player name to 1st place
    mov esi, OFFSET playerName
    mov edi, OFFSET topNames
    mov ecx, 16
    rep movsb                 ; Copy name
    mov BYTE PTR [topNames + 15], 0  ; Null terminate
    jmp UpdateDone

    ; === CHECK 2ND PLACE ===
Check2nd:
    cmp eax, topScores[4]     ; Is current score > 2nd place?
    jle Check3rd              ; No - check 3rd

    ; New 2nd place! Shift 2nd → 3rd
    mov ebx, topScores[4]
    mov topScores[8], ebx

    ; Copy name
    mov ecx, 0
CopyName1to2B:
    cmp ecx, 16
    jge DoneCopy1to2B
    mov bl, BYTE PTR [topNames + ecx + 16]
    mov BYTE PTR [topNames + ecx + 32], bl
    inc ecx
    jmp CopyName1to2B
DoneCopy1to2B:
    mov BYTE PTR [topNames + 32 + 15], 0

    ; Set new 2nd place
    mov topScores[4], eax

    ; Copy player name
    mov esi, OFFSET playerName
    mov edi, OFFSET topNames
    add edi, 16               ; Offset to 2nd slot
    mov ecx, 16
    rep movsb
    mov BYTE PTR [topNames + 16 + 15], 0
    jmp UpdateDone

    ; === CHECK 3RD PLACE ===
```

```
Check3rd:
    cmp eax, topScores[8]     ; Is current score > 3rd place?
    jle UpdateDone            ; No - not in top 3

    ; New 3rd place!
    mov topScores[8], eax

    ; Copy player name
    mov esi, OFFSET playerName
    mov edi, OFFSET topNames
    add edi, 32               ; Offset to 3rd slot
    mov ecx, 16
    rep movsb
    mov BYTE PTR [topNames + 32 + 15], 0

UpdateDone:
    popad
    ret
UpdateTopScores ENDP
```

Cascading Shift: When a new high score beats 1st place:

```
Before:
1st: 1000 (Alice)
2nd: 800 (Bob)
3rd: 600 (Charlie)

New score: 1200 (David)

After:
1st: 1200 (David)    ← New
2nd: 1000 (Alice)    ← Was 1st
3rd: 800 (Bob)       ← Was 2nd
(Charlie is dropped)
```

This ensures the top 3 are always sorted highest to lowest!

LoadScore - Loading High Scores

```
LoadScore PROC
    pushad

    ; Initialize arrays to zero (in case file doesn't exist)
    push edi
    push ecx
    push eax
```

```
    ; Clear scores
    mov edi, OFFSET topScores
    mov ecx, 3                ; 3 scores
    xor eax, eax              ; EAX = 0
ClearScores:
    mov DWORD PTR [edi], eax  ; Write 0
    add edi, 4                ; Next DWORD
    loop ClearScores

    ; Clear names
    mov edi, OFFSET topNames
    mov ecx, 48               ; 48 bytes
    xor al, al
    rep stosb                 ; Fill with zeros

    pop eax
    pop ecx
    pop edi

    ; Try to open file
    mov edx, OFFSET fileScore
    call OpenInputFile        ; Irvine32 procedure
    cmp eax, INVALID_HANDLE_VALUE
    je NoLoad                 ; File doesn't exist - use zeros

    mov fileHandle, eax

    ; Read 3 scores
    mov edx, OFFSET topScores
    mov ecx, 12
    call ReadFromFile

    ; Read 3 names
    mov edx, OFFSET topNames
    mov ecx, 48
    call ReadFromFile

    ; Close file
    mov eax, fileHandle
    call CloseFile

    ; Set highScore for compatibility
    mov eax, topScores[0]     ; 1st place = high score
    mov highScore, eax

NoLoad:
```

```
    popad
    ret
LoadScore ENDP
```

Defensive Initialization: We initialize arrays to zero BEFORE reading the file. If the file doesn't exist (first run), we have clean data rather than garbage!

SaveProgress / LoadProgress - Save States

The game saves progress so players can continue later:

```
SaveProgress PROC
    pushad

    ; Update saved data with current game state
    mov al, currentLevel
    mov savedLevel, al
    mov al, marioLives
    mov savedLives, al
    mov eax, score
    mov savedScore, eax
    mov al, marioX
    mov savedMarioX, al
    mov al, marioY
    mov savedMarioY, al
    mov al, marioPower
    mov savedPower, al
    mov al, coins
    mov savedCoins, al
    mov ax, timer
    mov savedTimer, ax

    ; Create file
    mov edx, OFFSET fileProgress  ; "mario_prog.dat"
    call CreateOutputFile
    cmp eax, INVALID_HANDLE_VALUE
    je NoProgSave

    mov fileHandle, eax

    ; Write 1 byte at a time (simplest approach)
    mov edx, OFFSET savedLevel
    mov ecx, 1
    call WriteToFile

    mov edx, OFFSET savedLives
    mov ecx, 1
```

```
    call WriteToFile

    mov edx, OFFSET savedScore
    mov ecx, 4                ; DWORD = 4 bytes
    call WriteToFile

    mov edx, OFFSET savedMarioX
    mov ecx, 1
    call WriteToFile

    mov edx, OFFSET savedMarioY
    mov ecx, 1
    call WriteToFile

    mov edx, OFFSET savedPower
    mov ecx, 1
    call WriteToFile

    mov edx, OFFSET savedCoins
    mov ecx, 1
    call WriteToFile

    mov edx, OFFSET savedTimer
    mov ecx, 2                ; WORD = 2 bytes
    call WriteToFile

    ; Close file
    mov eax, fileHandle
    call CloseFile

NoProgSave:
    popad
    ret
SaveProgress ENDP
```

Progress Data (Total: 12 bytes):

```
Byte 0:     Current Level (1 or 2)
Byte 1:     Lives remaining
Bytes 2-5:  Score (DWORD)
Byte 6:     Mario X position
Byte 7:     Mario Y position
Byte 8:     Mario power (0/1/2)
Byte 9:     Coins collected
Bytes 10-11: Timer remaining (WORD)
```

This lets players **pause mid-game, exit, and come back later** to the exact

same state!

Special Features

CheckFreezePowerup - Freeze All Enemies

```
CheckFreezePowerup PROC
    pushad

    cmp freezeCollected, 1    ; Already collected?
    je NoFreezeColl           ; Yes - skip

    ; Check if freeze exists
    mov al, freezeX
    or al, freezeY
    jz NoFreezeColl           ; Both zero = doesn't exist

    ; === CHECK X DISTANCE ===
    mov al, marioX
    sub al, freezeX           ; AL = difference
    cmp al, 1                 ; Within 1 tile?
    ja CheckFreezeNeg         ; No - check negative
    jmp CheckFreezeY

CheckFreezeNeg:
    cmp al, 255               ; -1 in unsigned (256-1)
    jne NoFreezeColl          ; Not close enough

CheckFreezeY:
    ; === CHECK Y DISTANCE ===
    mov al, marioY
    sub al, freezeY
    cmp al, 1
    ja CheckFreezeYNeg
    jmp FreezeHit

CheckFreezeYNeg:
    cmp al, 255
    jne NoFreezeColl

FreezeHit:
    ; === ACTIVATE FREEZE ===
    mov freezeCollected, 1
    mov freezeActive, 1
    mov freezeTimer, 150      ; 5 seconds at 30fps
```

```
    add score, 200            ; Bonus points

    ; Erase 'C' from screen
    mov eax, black + (black * 16)
    call SetTextColor
    mov dl, freezeX
    mov dh, freezeY
    inc dh
    call Gotoxy
    mov al, ' '
    call WriteChar

    ; Remove from map
    mov bl, freezeX
    mov bh, freezeY
    mov cl, ' '
    call SetTileAt

    ; Show message
    mov eax, lightCyan + (blue * 16)
    call SetTextColor
    mov dh, 0
    mov dl, 100
    call Gotoxy
    mov edx, OFFSET strFreeze ; "*** FREEZE ACTIVATED! ***"
    call WriteString
    call PlayFreezeSound

NoFreezeColl:
    popad
    ret
CheckFreezePowerup ENDP
```

Freeze Mechanic: When collected, enemies **stop moving** for 5 seconds. In `PlayGame`:

```
cmp freezeActive, 1
jne NoFreezeUpdate
cmp freezeTimer, 0
je FreezeEnd
dec freezeTimer           ; Count down
jmp SkipEnMove            ; DON'T call MoveEnemies!

FreezeEnd:
mov freezeActive, 0       ; Freeze expired
jmp SkipEnMove
```

```
NoFreezeUpdate:
; Normal gameplay - move enemies
mov al, frameCount
and al, 1
cmp al, 0
jne SkipEnMove
call MoveEnemies          ; Enemies move normally
```

This gives players a **strategic advantage** - 5 seconds to safely navigate dangerous areas!

ShootFireball - Projectile System

Only Fire Mario can shoot fireballs:

```
ShootFireball PROC
    pushad

    ; Find an inactive fireball slot
    xor ebx, ebx              ; EBX = index
FindFireSlot:
    cmp ebx, MAX_FIREBALLS    ; Checked all slots?
    jge NoFireSlot            ; Yes - all active, can't shoot
    cmp fireballActive[ebx], 0
    je FoundFireSlot          ; Found inactive slot!
    inc ebx
    jmp FindFireSlot

FoundFireSlot:
    ; Spawn fireball based on Mario's direction
    mov al, marioX
    cmp marioDir, 0           ; Facing left?
    je FireSpawnLeft

    inc al                    ; Spawn 1 tile to the right
    jmp FireSpawnDone

FireSpawnLeft:
    dec al                    ; Spawn 1 tile to the left

FireSpawnDone:
    mov fireballX[ebx], al
    mov al, marioY
    mov fireballY[ebx], al
    mov al, marioDir          ; Fireball moves in Mario's direction
    mov fireballDir[ebx], al
    mov fireballActive[ebx], 1
```

```
    call PlayFireballSound

NoFireSlot:
    popad
    ret
ShootFireball ENDP
```

Spawn Offset: Fireball spawns **1 tile ahead** of Mario, not at Mario's position. This prevents instant self-collision!

MoveFireballs - Projectile Physics

```
MoveFireballs PROC
    pushad

    xor ebx, ebx              ; EBX = fireball index
FireLoop:
    cmp ebx, MAX_FIREBALLS
    jge FireDone
    cmp fireballActive[ebx], 0
    je NextFire

    ; Erase old position
    push ebx
    mov eax, black + (black * 16)
    call SetTextColor
    mov dl, fireballX[ebx]
    mov dh, fireballY[ebx]
    inc dh
    call Gotoxy
    mov al, ' '
    call WriteChar
    pop ebx

    ; Move fireball
    cmp fireballDir[ebx], 0   ; Moving left?
    je FireLeft
    inc fireballX[ebx]        ; Move right
    jmp CheckFireColl
FireLeft:
    dec fireballX[ebx]        ; Move left

CheckFireColl:
    ; Check bounds
    movzx eax, fireballX[ebx]
    cmp eax, 2
```

```
    jle DeactiveFire          ; Hit left edge
    cmp eax, 117
    jge DeactiveFire          ; Hit right edge

    ; Check wall collision
    push ebx
    mov bl, fireballX[ebx]
    mov bh, fireballY[ebx]
    call GetTileAt
    call IsBlockingTile
    pop ebx
    jz DeactiveFire           ; Hit wall

    ; Check enemy collision
    call CheckFireballEnemyHit
    jmp NextFire

DeactiveFire:
    mov fireballActive[ebx], 0

NextFire:
    inc ebx
    jmp FireLoop

FireDone:
    popad
    ret
MoveFireballs ENDP
```

Fireball Speed: Fireballs move **1 pixel per frame** (same speed as Mario's horizontal movement). This feels balanced - fast enough to be useful, slow enough to be dodgeable by enemies (if we added enemy AI).

CheckFireballEnemyHit - Projectile Collision

```
CheckFireballEnemyHit PROC
    ; Input: EBX = fireball index
    pushad

    movzx ecx, enemyCount
    cmp ecx, 0
    je NoFireEnHit

    xor edi, edi              ; EDI = enemy index

FireEnLoop:
```

```
    cmp enemyActive[edi], 0
    je NextFireEn

    ; Check exact position match
    mov al, fireballX[ebx]
    cmp al, enemyX[edi]
    jne NextFireEn            ; X doesn't match

    mov al, fireballY[ebx]
    cmp al, enemyY[edi]
    jne NextFireEn            ; Y doesn't match

    ; === HIT! ===
    mov enemyActive[edi], 0   ; Kill enemy
    mov fireballActive[ebx], 0 ; Destroy fireball
    add score, 150            ; More points than stomp!
    call PlayStompSound

    ; Erase enemy
    push ebx
    mov eax, black + (black * 16)
    call SetTextColor
    mov dl, enemyX[edi]
    mov dh, enemyY[edi]
    inc dh
    call Gotoxy
    mov al, ' '
    call WriteChar
    pop ebx
    jmp NoFireEnHit           ; One fireball can only hit one enemy

NextFireEn:
    inc edi
    cmp edi, ecx
    jl FireEnLoop

NoFireEnHit:
    popad
    ret
CheckFireballEnemyHit ENDP
```

Collision Detection: We check for **exact match** (same X and Y). This is stricter than coin/enemy collision with Mario (which allows 1-tile tolerance). Makes fireballs require **more precision** to aim!

Performance Optimizations

Frame-Rate Management

The game maintains a consistent **30 FPS** through careful timing:

```
GameTick:
    ; ... game logic ...
    call RenderFrame

    mov eax, 45               ; 45 milliseconds
    call Delay                ; Wait

    ; 1000ms / 45ms   22 FPS
    ; But processing time adds ~8-10ms per frame
    ; Total: 45 + 8 = 53ms   19 FPS (close enough)

    jmp GameTick
```

In practice, the game runs at approximately **19-22 FPS** depending on CPU speed. This is acceptable for a turn-based platformer!

Incremental Rendering

Full screen redraws are expensive:

```
120 columns × 27 rows = 3,240 characters
Each character requires:
- Gotoxy call (set cursor position)
- SetTextColor call
- WriteChar call
Total: ~9,720 function calls per full redraw!
```

Incremental rendering only updates what changed:

```
- HUD (120 characters)
- Mario (1 character)
- ~5 enemies (5 characters)
- ~10 coins (10 characters)
- ~3 fireballs (3 characters)
Total: ~139 characters = **96% reduction!**
```

Entity Activation Flags

Instead of checking every slot in arrays, we use `active` flags:

```
; SLOW:
mov ecx, MAX_ENEMIES          ; Check all 10 slots
CheckAllEnemies:
    ; Process enemy even if doesn't exist
```

```
    dec ecx
    jnz CheckAllEnemies

; FAST:
movzx ecx, enemyCount         ; Only check active enemies
CheckActiveEnemies:
    cmp enemyActive[ebx], 0
    je SkipThisOne            ; Skip inactive
    ; Process active enemy
SkipThisOne:
    inc ebx
    loop CheckActiveEnemies
```

If only 3 enemies are active, we skip 7 unnecessary checks!

Game Balance & Design

Timer System

Each level has a **150-second timer**:

```
mov al, frameCount
and al, 31                ; Check every 32 frames (~1 second)
cmp al, 0
jne NoTimerDec
cmp timer, 0
je TimeDie                ; Time's up - death!
dec timer                 ; Decrease timer
```

Time Bonus: Remaining time is converted to points:

```
; Level complete
movzx eax, timer
shl eax, 1                ; Multiply by 2
add score, eax            ; Bonus points!

; Victory (beat entire game)
movzx eax, timer
shl eax, 2                ; Multiply by 4 (double bonus!)
add score, eax
```

This encourages **speed runs** - finish faster for more points!

Power-Up Progression

The power system creates a clear **progression**:

1. **Small Mario** (`m`)

- Dies in one hit
- Can't shoot fireballs
- Vulnerable

2. **Big Mario** (`M`)
   - Takes one hit, downgrade to small
   - Still can't shoot
   - More resilient
3. **Fire Mario** (`M` white on red)
   - Takes one hit, downgrade to big
   - Can shoot fireballs
   - Most powerful

Invincibility Frames: After taking damage, 2 seconds of invincibility prevents instant death from continuous collision. Visual feedback: Mario flashes yellow/magenta!

Score Requirements

To prevent "rushing through," levels require minimum scores: - **Level 1:** Need 700+ score - **Level 2:** Need 500+ score

If you reach the flag without enough score, you're sent back to collect more coins/defeat enemies!

Conclusion

This Super Mario Bros clone demonstrates many fundamental game programming concepts:

1. **State Machines** - Clean separation of menu/gameplay/pause/gameover
2. **Physics Simulation** - Gravity, velocity, collision
3. **Entity Systems** - Parallel arrays for enemies/coins/fireballs
4. **Rendering Optimization** - Incremental updates, dirty flags
5. **File I/O** - Persistent high scores and save states
6. **Input Handling** - Non-blocking keyboard input
7. **Sound Effects** - MP3 playback via Windows MCI
8. **Collision Detection** - Tile-based and entity-based
9. **Game Balance** - Timer pressure, score requirements, power-up progression

All implemented in **x86 Assembly Language** - one of the most challenging but educational ways to create a game!

Total Lines of Code: ~2,300 lines **Total Procedures:** 50+ **Features:** 2 levels, 3 enemy types, 5 power-up types, freeze mechanic, double jump, breakable platforms, mystery blocks, save system, high score tracking, timer bonus, invincibility frames

---

For Your Demo

Key Points to Highlight:

1. **Sound System** - Show MP3 playback via MCI, explain fallback to beeps
2. **Physics Engine** - Demonstrate gravity, jumping, double-jump
3. **Power-Up System** - Show progression: small→big→fire
4. **Special Features** - Freeze power, breakable platforms, mystery blocks
5. **Save System** - Show continue game, high scores
6. **Enemy AI** - Simple but effective patrol behavior
7. **Collision Detection** - Explain stomp vs side collision
8. **File Format** - Show how data is stored (binary)
9. **Rendering** - Explain incremental vs full redraw
10. **Assembly Techniques** - REP MOVSB, MOVSX, parallel arrays

Live Demo Suggestions: - Play through level 1, collect coins, stomp enemies - Hit mystery block to spawn mushroom - Collect freeze power and watch enemies stop - Shoot fireballs as Fire Mario - Die and show respawn mechanic - Complete level to show time bonus - Check high scores screen

Good luck with your demo!