Team7 BananaSlug Documentation

Contents

1 CPU Registers

1.1 Register File Registers

The Banana CPU has a Register File[1] with 32 addressable 16-bit registers. They are indexed with `reg_a`, `reg_b`, `reg_c` as seen in CPU Instructions. The Zero Register and Stack Pointer Register serve special purposes as defined below.

1.1.1 Zero Register (`registers[0]`)

The Zero Register is a special Register File register at `registers[0]` that always returns the value zero, and is uneffected by attempted writes.

1.1.2 Stack Pointer Register (`registers[29]`)

The stack is located at memory locations 0x3200-0x51ff, (see Address Space). The Stack Pointer Register is a special Register File register at `registers[29]` which shall be initialized to 0x5200 on a Reset Sequence.

The Stack Pointer is expected to return to its initial value (0x5200) after a call to either `loop()` or `setup()`.

The stack works top-down, so when a word is pushed onto the stack, the stack pointer is decreased. When a word is pulled from the stack, the stack pointer is increased. If the stack pointer ever reaches a value outside the stack memory, exection shall stop.

1.2 Program Counter

The Program Counter (PC) is a 16-bit register outside of the Register File which holds the address of the next instruction to be executed. As instructions are executed, the value of the program counter is updated, usually moving on to the next instruction in the sequence. The value can be affected by branch and jump instructions.

The Program Counter is only valid for values greater than 0x8000. When running the Operating System, the Program Counter shall be held at 0x0000. A call to `loop()` or `setup()` is done by first setting the Program Counter to 0xfffc, then running a `JAL` to the desired function address, (see SLUG Address Space). Note that after the successful completion of a call to `loop()` or `setup()`, the Program Counter will return to 0x0000.

[1] A Register File is block of memory representing an addressable array of registers.

2 CPU Instructions

All CPU Instructions are 4-bytes long, and have an opcode in their top 6 bits. The opcode must be used to decode the rest of the instruction according to what type of instruction it is: I-Type or R-Type. After each instruction, the Program Counter Register is increased by 4 unless executing a branch or jump instruction.

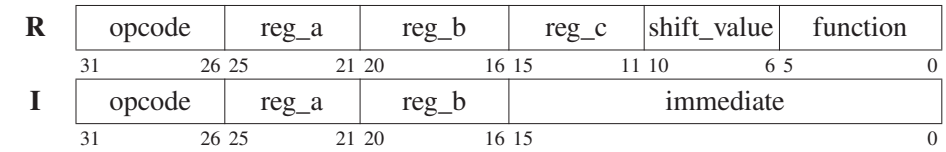

Figure 1: Instruction Formats

Any unknown instructions shall be treated to mean "No Operation", or `NOP`. A `NOP` simply increments the PC by 4, without affecting any other memory or register.

2.1 I-Type Instructions

2.1.1 Branch On Equal (`BEQ`)

- **Opcode**: 2
- **Operation**: `if(R[reg_a]==R[reg_b]) PC=PC+4+ 4*Immediate`

2.1.2 Store Byte (`SB`)

- **Opcode**: 11
- **Operation**: `M[R[reg_a]+Immediate] = R[reg_b](7:0)`

2.1.3 Jump And Link (`JAL`)

- **Opcode**: 24
- **Operation**: `R[31]=PC+4; PC=4*Immediate`

2.1.4 Load Byte Unsigned (`LBU`)

- **Opcode**: 25
- **Operation**: `R[reg_b]=M[R[reg_a]+Immediate](7:0)`

2.1.5 Jump (`J`)

- **Opcode**: 37
- **Operation**: `PC=4*Immediate`

2.1.6 Add Immediate (ADDI)

- **Opcode**: 46
- **Operation**: `R[reg_b] = R[reg_a] + Immediate`

2.1.7 Branch On Not Equal (BNE)

- **Opcode**: 51
- **Operation**: `if(R[reg_a]!=R[reg_b]) PC=PC+4+ 4*Immediate`

2.1.8 Load Word (LW)

- **Opcode**: 53
- **Operation**: `R[reg_b] = M[R[reg_a]+Immediate]`

2.1.9 Store Word (SW)

- **Opcode**: 58
- **Operation**: `M[R[reg_a]+Immediate] = R[reg_b]`

2.2 R-Type Instructions

2.2.1 Subtract (SUB)

- **Opcode**: 0
- **Function**: 0
- **Operation**: `R[reg_c] = R[reg_a] - R[reg_b]`

2.2.2 Shift Right Logical (SRL)

- **Opcode**: 0
- **Function**: 13
- **Operation**: `R[reg_c] = (unsigned) R[reg_b] >> shift_value`

2.2.3 And (AND)

- **Opcode**: 0
- **Function**: 19
- **Operation**: `R[reg_c] = R[reg_a] & R[reg_b]`

2.2.4 Nor (NOR)

- **Opcode**: 0
- **Function**: 21
- **Operation**: `R[reg_c] = ~(R[reg_a] | R[reg_b])`

2.2.5 Shift Right Arithmetic (SRA)

- **Opcode**: 0
- **Function**: 25
- **Operation**: `R[reg_c] = (signed) R[reg_b] >> shift_value`

2.2.6 Shift Left Logical (SLL)

- **Opcode**: 0
- **Function**: 30
- **Operation**: `R[reg_c] = R[reg_b] << shift_value`

2.2.7 Jump Register (JR)

- **Opcode**: 0
- **Function**: 40
- **Operation**: `PC=R[reg_a]`

2.2.8 Or (OR)

- **Opcode**: 0
- **Function**: 50
- **Operation**: `R[reg_c] = R[reg_a] | R[reg_b]`

2.2.9 Set Less Than (SLT)

- **Opcode**: 0
- **Function**: 57
- **Operation**: `R[reg_c] = (R[reg_a] < R[reg_b])`

2.2.10 Add (ADD)

- **Opcode**: 0
- **Function**: 60
- **Operation**: `R[reg_c] = R[reg_a] + R[reg_b]`

3 Address Space

All data is big-endian.

3.1 Permissions

Permissions are handled as follows:

- "r" indicates a readable address range where load instructions succeed if the load address is readable, and recieve 0 is the load address is not readable.
- "w" denotes a writable address range where store instructions succeed if the store address is writable, and act as `NOP` if the store address is not writable.
- "x" signifies an executable address range where instructions are run if the PC address is executable, and act as `NOP` if the PC address is not executable.

3.2 RAM Address Space

Address	Size (bytes)	Permissions	Contents
0x0000	0x7000	rw	RAM
0x3200	0x2000	rw	Stack
0x5200	0x1e00	rw	VRAM

3.3 IO Address Space

See IO for functions of each address.

Address	Size (bytes)	Permissions	Contents
0x7000	1	r	Controller Data
0x7100	1	r	Debug `stdin`
0x7110	1	w	Debug `stdout`
0x7120	1	w	Debug `stderr`
0x7200	1	w	Stop Execution

3.4 SLUG Address Space

Address	Size (bytes)	Permissions	Contents
0x8000	0x8000	rx	SLUG File
0x8000	4	rx	`"SLUG"`
0x81e0	4	rx	Address to `setup()`
0x81e4	4	rx	Address to `loop()`
0x81e8	4	rx	Load Data Address (ROM)
0x81ec	4	rx	Program Data Address (RAM)
0x81f0	4	rx	Data Size

The contents of the SLUG file header are used by the Operating System.

4 IO

4.1 Game Input IO

- Reading 1 byte from `0x7000` shall read the current state of the game controller.

Bit	7	6	5	4	3	2	1	0
Button	A	B	SELECT	START	UP	DOWN	LEFT	RIGHT

```
#define CONTROLLER_A_MASK      ((uint8_t)0x80)
#define CONTROLLER_B_MASK      ((uint8_t)0x40)
#define CONTROLLER_SELECT_MASK ((uint8_t)0x20)
#define CONTROLLER_START_MASK  ((uint8_t)0x10)
#define CONTROLLER_UP_MASK     ((uint8_t)0x08)
#define CONTROLLER_DOWN_MASK   ((uint8_t)0x04)
#define CONTROLLER_LEFT_MASK   ((uint8_t)0x02)
#define CONTROLLER_RIGHT_MASK  ((uint8_t)0x01)
```

Figure 2: Banana Controller Diagram

4.2 Debug IO

- Reading from `0x7100` shall request 1 byte from the debug terminal's `stdin`.
- Writing to `0x7110` shall write 1 byte to the debug terminal's `stdout`.
- Writing to `0x7120` shall write 1 byte to the debug terminal's `stderr`.
- Writing anything to `0x7200` shall stop Banana execution.

5 Operating System

The Banana Operating System is responsible for memory management and Slug file execution. Before reading this section, be sure to understand the Banana address space, (See Address Space).

5.1 Reset Sequence

The Reset Sequence shall be run only once, on start-up.

1. Clear all of RAM with zeros
2. Copy `data` section to RAM
3. Initialize stack pointer register to the end of the stack (0x5200)
4. Call `setup()`
5. Begin Game Loop Sequence

5.2 Game Loop Sequence

The Game Loop Sequence shall be run repeatedly, at a target frequency of 60 Hz. (Commonly referred to as 60 frames per second, or FPS). If an iteration of a Game Loop Sequence completes sooner than the target period (~16.667 ms), then there shall be additional delay added before starting the following Game Loop Sequence. If an iteration of a Game Loop Sequence takes longer than the target period (~16.667 ms), then that Game Loop Sequence is completed as normal, then the following Game Loop Sequence begins with no additional delay.

1. Run `loop()`
2. The GPU frame buffer is displayed

6 GPU

The Graphics Processing Unit (GPU) is responsible for decoding the Video RAM (VRAM) and rendering its contents onto the screen.

6.1 VRAM and Frame Buffer

The GPU only has access to the VRAM address space (0x5200-0x6fff). The VRAM holds a single frame buffer that includes all the data for drawing a single frame.

The display resolution is defined as 64 pixels in width and 60 pixels in height. The pixel address can be calculated as follows:

```
int getPixelAddress(int width, int height) {
    int pixel_index = width + (height * 64);
    int pixel_offset = 2 * pixel_index;
    return 0x5200 + pixel_offset;
}
```

6.2 Pixel Encoding

Each pixel is encoded as 2 bytes with levels for red, green, and blue each encoded with 5 bits.

Bit	Color
15	Unused
14-10	Red (0x00:no-red. . . 0x1f:full-red)
9-5	Green (0x00:no-green. . . 0x1f:full-green)
4-0	Blue (0x00:no-blue. . . 0x1f:full-blue)

Examples:

- Yellow: 0x7fe0
- Cyan: 0x03ff
- Magenta: 0x7c1f

6.3 Rendering

Upon completion of each `loop()` iteration by the CPU, the GPU decodes and displays the contents of the frame buffer.

7 References

The Banana Controller Diagram used in this work as seen in Figure 2 is derived from a work by Fant0men and is licensed under the Creative Commons Attribution-Share Alike 3.0 Unported license. Modifications have been made to the original work. To view the original work, visit https://en.wikipedia.org/wiki/File:Nes_controller.svg.