SABENE SPECIFICATION

Created by Jacob
12-06-2023
17:41

[Insert Emulator Architecture Picture Here]

Table of Contents

1 – What is Sabene?

Sabene is an 8-bit emulator written entirely in C. The emulator itself has drivers that allow communication between the emulated GPU, ROM disk, VGA, Keyboard, and Speaker.

The emulator runs a custom CPU and GPU architecture that is only seen in the Sabene emulator. This means that x86-64, ARM, or other types of assembly architecture will *not* run on Sabene emulators. You *must* assemble and create programs with the assembler packaged with your *Sabene Emulator*.

1.1 – Why use Sabene?

Sabene is a fun little project created in order to explore the ins-and-outs of computer architectures, emulation, and driver emulation. Users of sabene can write their own programs utilizing the assembler packaged with the copy of *Sabene Emulator* that they download from the github: *https://github.com/jaythom2723/sabene-8*. Programmers can use Sabene to make games, produce software, and potentially even operating systems. The limits of Sabene are limited only to the programmer's creativity and skill.

1.2 – What is Sabene used for?

Sabene can be used for general algorithm design, game development, driver development, os development, and more. The emulator itself is limited with its assembly instruction set, so programming might opt to use the proprietary Sabene programming language. The Sabene programming language makes it easier for programmers to interface with memory directly, define variables, call functions, as well as create applications easier.

1.3 – About the Developer

My name is Jacob, I am an aspiring software developer who focuses in C. I am currently learning NASMx86_64 assembly, computer architectures, Godot, and OpenGL. I am actively researching how to create device drivers using assembly and C, however, am currently struggling. This emulator is my magnum opus, a combination of everything I've learned throughout my years of creating emulators. Despite being only a few years into learning the quirks of C-style programming, I can comfortable say that I've mastered the language for application-based programs.

2 – The CPU

2.1 – Registers

The CPU architecture within *all* Sabene Emulators is the JS816A “Jacob’s Sabene 8-bit cpu with 16-bit Address”. The CPU has 8, 8-bit general purpose registers, 1 4-bit flag register, 1 8-bit overflow register, 1 8-bit stack pointer, and 1 8-bit program counter. There are 4 extra registers specifically for floating point values. All of these registers are 8-bit as well. The CPU also has 1 3-bit Bus Interface Register.

Register	Size (bits)	Purpose
R00	8	General Purpose
R01	8	General Purpose
R02	8	General Purpose
R03	8	General Purpose
R04	8	General Purpose
R05	8	General Purpose
R06	8	General Purpose
R07	8	General Purpose
R08	4	Flag Register
R09	8	Overflow Register
R10	8	Stack Pointer
R11	8	Program Counter
F01	8	General Purpose Floating Point
F02	8	General Purpose Floating Point
F03	8	General Purpose Floating Point
F04	8	General Purpose Floating Point
BIR	3	Bus Interface Register

2.2 – Bus Interface Register

The CPU can communicate between the ROM disk, RAM, as well as the GPU through a bus interface. The CPU can control where data is sent directly with the Bus Interface Register.

Name of Value	Bit 3	Bit 2	Bit 1
R/W RAM	0	0	1
R ROM	0	1	0
R/W VRAM	0	1	1
R keyboard	1	0	0
R/W disk	1	0	1
R/W VGA color table	1	1	0
R mouse	1	1	1

The bus interface register can be set via the following instruction: `SBR <value>` The “Set Bus Register” instruction will set the value of the cpu’s Bus Interface Register. The assembler will automatically set the Bus Interface Register if the 16-bit memory address passed into an operation enters its range. For instance, to send data to the GPU: `STR r0, $2000` The following S8 assembly code will be expanded on during assembly into the following code:

```
BIR b011                                        ; Set BIR to 3 (send to VRAM)
STR r0, $0                                      ; Store R0 at VRAM[0]
BIR b001                                        ; Set BIR to 1 (RAM)
```

The BIR value will also automatically reset after the last address in range of a hardware's memory addresses. For instance:

```
; Before assembly
STR r0, $2000
STR r0, $2001
STR r0, $2002
; After assembly
BIR b011                                        ; Set BIR to 3 (send to VRAM)
STR r0, $0                                      ; Store R0 at VRAM[0]
STR r0, $1                                ; Store R0 at VRAM[1]
STR r0, $2                                ; Store R0 at VRAM[2]
BIR b000                                        ; Set BIR to 0 (don't send)
```

2.3 – Arithmetic Logic Unit

The Arithmetic Logic Unit (ALU) is the chip that handles *all* things math-related. The ALU can handle floating point values as well as negative numbers. The ALU can take in any general purpose register's value as well as memory addresses. You can add, subtract, divide, multiply, as well as use bitwise and, or, and xor operation. The arithmetic logic unit also allows for a `CMP <value>,<value/address>`. See 2.3.1 for more information on CMP.

```
MOV r0, 2                                 ; r0 = 2
MOV r1, 2                                 ; r1 = 2
ADD r0, r1                                ; r0 + r1 = 4 = r0
SUB r0, r1                                ; r0 – r1 = 2 = r0
MUL r0, r1                                ; r0 * r1 = 4 = r0
DIV r0, r1                                ; r0 / r1 = 2 = r0
AND r0, r1                                ; r0 & r1 = 2 = r0
XOR r0, r0                                ; r0 ^ r0 = 0 = r0
OR  r0, r1                                ; r0 | r1 = 2 = r0
```

2.3.1 – Compare & Conditionals

Conditionals are handled by the ALU and set the 4-bit flag register based on whatever the value of the check is. The CMP instruction offers an easier way to check values against one another, however, the bitwise operators can also be used to check for conditions. A good example of this would be in Chapter TBD.

```
MOV r0, 0                                       ; r0 = 0
MOV r1, 2                                       ; r1 = 2
CMP r0, r1                                      ; r0 ? r1
JEQ $#                                          ; if(r0 == r1) JMP $#
```

2.4 – Read/Write RAM

The CPU, by default, will read and write data to and form Random-Access-Memory (RAM). The CPU refers to memory throug a 16-bit address line that it can send memory addresses to and receive that data back. Storing data into memory is done with the STR instruction; reading data from memory is done with the RD instruction. For instance:

```
MOV r0, 123                                           ; r0 = 123
STR r0, $0000                                         ; Store r0 at RAM[0]
RD  r1, $0000                                         ; Read RAM[0], r1 = RAM[0]
```

The CPU stores and reads values from memory in little endian, similar to the *Intel* and *AMD* architectures.

3 – Pseudo-instructions

3.1 – What are Pseudo-instructions?

Pseudo-instructions, or “directives”, are “fake” instructions that are only used by the assembler to give clues to the CPU about what to do with a piece of data. For instance, the “db” directive defines N bytes into data memory. However, the CPU does not openly contain an opcode for: “db”.

3.2 – Defining constants with “DB” and friends

The assembler has multiple directives for defining data into memory. The following instructions the define data into memory are the following instructions:

```
DB                              ; Define Byte – Define N bytes to data memory
DW                              ; Define Word – Define N words to data memory
DF                              ; Define Float – Define N floats to data memory
```

Defining multiple bytes/words/floats in memory at a time is supported by comma separated list, offering for sequential string defintions, unicode characters, and much more. Words and floats are stored into memory via little endian.

3.3 – Reserving bytes with “RESB” and friends

The assembler also has multiple directives for *reserving* data into memory, which means the data is not constant and can be modified at runtime of the application. The instructions for reserving data into memory are the following instructions:

```
RESB                            ; Reserve Byte – Reserve N bytes to data memory
RESW                            ; Reserve Word – Reserve N words to data memory
RESF                            ; Reserve Float – Reserve N floats to data memory
```

Reserving multiple bytes/words/floats into data memory is also allowed via comma-separation, just like the DB/DW/DF directives.

4 – Constants

Constants are the most common and prevalent forms of data in a computer. They can be set via the MOV instruction, the EQU directive, or the DB/DW/DF directives.

4.1 – Integrals

```
DB  00h                                  ; hexadecimal constant
DB  16                                   ; integer constant
DW  0000h                                ; hexadecimal word constant
DW  65535                                ; integer word constant
EQU 12                                   ; integer byte constant
EQU 13h                                  ; hexadecimal byte constant
MOV r0, 12                               ; r0 = 12
MOV r0, 12h                              ; r0 = 12h hexadecimal!
```

4.2 – Floating Point

Floating point values can only be set by moving their value into a floating point register. These registers are the following:

```
MOV f01, 0.12                            ; f01 = 0.12
MOV f02, 0.34                            ; f02 = 0.34
```

4.3 – Characters

Character values can be set by using DB or MOV.

```
DB ‘H’
DB ‘H’, ‘E’, ‘L’, ‘L’, ‘O’
MOV r0, ‘h’
```

4.4 – Strings

String values can be set using DB.

```
DB “Hello, World!”, 13, 10, 0
```