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 programmers 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. More on the sabene programming language in Chapter [TBD].

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 comfortably 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	6	Flag Register
R09	8	Overflow Register
R10	8	Stack Pointer
R11	8	Program Counter
F00	8	GP Floating Point Register
F01	8	GP Floating Point Register
F02	8	GP Floating Point Register
F03	8	GP Floating Point Register
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 for conditionals with bitwise operators can be found below:

```
MOV r0, 0                                   ; r0 = 0
MOV r1, 2                                   ; r1 = 2
CMP r0, r1                                  ; r0 ? r1
JEQ $#                                      ; if(r0 == r1) JMP $#
;     BITWISE CONDITIONAL
MOV R0, 4                                   ; r0 = 4
MOV R1, 5                                   ; r1 = 5
XOR R0, R1                                  ; r0 ^ r1 = r0
JNZ $@                                      ; if(r0 != r1 /*true*/) JMP $#
```

2.4 – Read/Write RAM

The CPU, by default, will read and write data to and from Random-Access-Memory (RAM). The CPU refers to memory through a 16-bit address line. 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], RAM[0] = 123
RD  r1, $0000                               ; Read RAM[0], r1 = RAM[0] = 123
```

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

2.5 – Interrupts

2.5.1 – What is an interrupt?

An interrupt is a way for the CPU to stop processing a program and begin processing something else for a brief period of time. Interrupts allow *Sabene Emulator*'s to read and write from the various pieces of hardware located inside of *Sabene*.

2.5.2 – Interrupts explained

Interrupts, as stated in 2.5.1, are an event that alters the processing sequence of the CPU. These interrupts can be planned or unplanned and will always return to the original process after their processing is complete. For instance:

```
BIR b100                        ; Read Keyboard
MOV r0, $0000                   ; Read single character
INT                             ; Wait for keyboard input
BIR b001                        ; Reset bus interface register
```

2.5.3 – Interrupt Table

BIR value (bits)	INT name	INT description
100	RKEY	Read from Keyboard
101	RDISK	Read from Disk
101	WDISK	Write to Disk

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
```

5 – Memory

5.1 – Memory Overview

Memory is not a solid chunk of addresses or data. Memory in Sabene-8, similar to other systems, is segmented. These segments contain varying amounts of data of varying types and names. The segments are as follows:

Segment Name	Segment Base	Segment Limit
Boot Segment	$0000	$0200
Data Segment	$0201	$0FFF
Program Segment	$1000	$1FFF
Video Memory Segment	$2000	$6B00
Disk Segment	$6B01	$7B00
Stack Segment	$7B01	$8B00
VGA Color Segment	$8B01	$D601
Video Data Segment	$D602	$E602
Free Memory Segment	$E603	$FFFF

5.2 – Referencing Memory

Memory addresses are referred in assembly with the following method:

```
STR r0, $0100     ; Store r0 into RAM[256], $0100 is located in the boot segment
```

When referring to a memory address, if the value lies within a given segment that does not refer to RAM, the assembler will auto-expand the store instruction to give clues to the CPU about where a memory address lies. This is discussed earlier in Chapter 2.2.

Anytime an address is referred to in assembly code, the assembly will automatically replace that value with the address of the label, unless that address has been dereferenced. In which case, the assembler will grab the *value* stored in memory and supply it as opposed to the address. For example:

```
MSG: DB “Hello, World!”, 13, 10, 0  ; Define a MSG label as a set of defined bytes
MOV r10, MSG                        ; Set the stack pointer to the MSG address
```

5.3 – Dereferencing Memory

Memory addresses can be dereferenced by surrounding an address or label in brackets `[]`. Dereferencing memory like this simply tells the assembler to replace the address provided into the value at that memory location. Relatively straight forward. Memory addresses can be incremented for precise access to memory and the dereferencing provided even more direct memory management. For instance:

```
MSG: DB “Hello, World!”, 13, 10, 0  ; Define a MSG label as a set of defined bytes
MOV r0, [MSG]                       ; r0 = *msg; (C equivalent)
```

6 – Video

6.1 – Graphic’s Processing Unit (GPU)

The Graphic’s Processing Unit (GPU) is where all graphical calculations or offloaded processes are handled. Offloaded processes from the CPU will be stored into Video Random Access Memory (VRAM), before the internal Video Program Counter Register (VPCR) takes over the process.

Processes that run on the CPU are completely independent from GPU processes and these two processes *must* communicate their data to each other in order to prevent memory leaks. See Chapter 6.4 for more information on Interprocess Communication.

The GPU also has 2 cores, allowing for multiple threads of execution on the GPU chip itself. See chapter 6.5, *GPU Cores*, for more information on GPU Cores.

6.2 – Video Random Access Memory (VRAM)

Video Random Access Memory (VRAM) is how the GPU stores information. The GPU, similar to the CPU, can read and write to VRAM on demand. Not only that, but VRAM, similar to RAM, is segmented. The memory segments are listed in Chapter 5.1, *Memory Overview*, however, will be listed below.

Segment Name	Segment Base	Segment Limit
Video Memory Segment	$2000	$6B00
VGA Color Segment	$8B01	$D601
Video Data Segment	$D602	$E602

6.3 – Video Instruction Subset

The Video Instruction Subset is currently in development.

6.4 – Interprocess Communication

When a process first becomes offloaded from the CPU, the GPU will know nothing about the previous variables defined into RAM. Instead, it will rely on what has been provided into VRAM by the processor. The values passed into VRAM will act as “parameters” to the processor to be run on the GPU. If these values are not passed into VRAM prior to the process being offloaded, then the process could fail and cause catastrophic damage to the *Sabene* system.

The GPU will also need to send the data *back* to the CPU, as they do not communicate with each other by default and act completely independently from each other.

7 – Video Graphics Array (VGA)

The Video Graphics Array is the chosen platform for displaying colors onto the screen. The VGA controller chip in the *Sabene Emulator* supports, by default, 15 colors. However, a bit flag can be set by the internal Graphics Processing Unit (GPU), seen in Chapter 6. This bit flag must be sent to the first memory location in video memory. The value will be reset after the bit flag is set and *must* be set during the emulator’s boot-up process.

7.1 – VGA Color Lookup Table

The VGA Color Table is a *lookup-table* that the GPU uses to color pixels on the screen. The values of colors are specified below:

8 – Audio

TBD

9 – Keyboard

Reading from the keyboard is done by setting the `bir` register. The `bir` register must be set to the corresponding “Read Keyboard” value, see Chapter 2.2 for more information on setting the `bir` register.

After setting the `bir` register, the following assembly code will wait for a keypress before continuing:

```
BIR b100                                    ; Read from keyboard
MOV r0, $0000                               ; Read single character
INT                                         ; Interrupt CPU (See Chapter 2.5)
```

10 – Assembly Specification

10.1 -

10.2 – Instruction Table

Ins	# Args	Opcode (hex)	Opcode (bits)	Size (bytes)
MOV	2	01	FF 000001	3
STR	2	02	FF 000010	3
RD	2	03	FF 000011	3
ADD	2	04	FF 000100	3
SUB	2	05	FF 000101	3
DIV	2	06	FF 000110	3
MUL	2	07	FF 000111	3
AND	2	08	FF 001000	3
XOR	2	09	FF 001001	3
OR	2	0A	FF 001010	3
CMP	2	0B	FF 001011	3
JMP	1	0C	FF 001100	3
JEQ	1	0D	FF 001101	3
JNQ	1	0E	FF 001110	3
JC	1	0F	FF 001111	3
JNC	1	10	FF 010000	3
JL	1	11	FF 010001	3
JNL	1	12	FF 010010	3
JLE	1	13	FF 010011	3
JNLE	1	14	FF 010100	3
JG	1	15	FF 010101	3
JNG	1	16	FF 010110	3
JGE	1	17	FF 010111	3
JNGE	1	18	FF 011000	3
INT	0	19	FF 011001	3
PUSH	1	1A	FF 011010	2
POP	1	1B	FF 011011	2
CALL	1	1C	FF 011100	3
RET	0	1D	FF 011101	1
HLT	0	1E	FF 011110	1
BIR	1	1F	FF 011111	2
INC	1	20	FF 100000	2
DEC	1	21	FF 100001	2
PUSHA	0	22	FF 100010	1
CALLZ	1	23	FF 100011	3
CALLNZ	1	24	FF 100100	3
CALLC	1	25	FF 100101	3
CALLNC	1	26	FF 100110	3
CALLL	1	27	FF 100111	3
CALLNL	1	28	FF 101000	3
CALLLE	1	29	FF 101001	3

CALLNLE	1	2A	FF 101010	3
CALLG	1	2B	FF 101011	3
CALLNG	1	2C	FF 101100	3
CALLGE	1	2D	FF 101101	3
CALLNGE	1	2E	FF 101110	3
CMVZ	2	2F	FF 101111	3
CMVNZ	2	30	FF 110000	3
CMVC	2	31	FF 110001	3
CMVNC	2	32	FF 110010	3
CMVL	2	33	FF 110011	3
CMVNL	2	34	FF 110100	3
CMVLE	2	35	FF 110101	3
CMVNLE	2	36	FF 110110	3
CMVG	2	37	FF 110111	3
CMVNG	2	38	FF 111000	3
CMVGE	2	39	FF 111001	3
CMVNGE	2	3A	FF 111010	3