x595g Guide (Version 1.5)

Colin Unger

February 24, 2020

Contents

Chapter 1

Forward

1.1 Terminology

by7e: 7 bits

Chapter 2

Basics

2.1 Registers

There are 11 total registers. Each is 28 bits in size.
There are eight general purpose registers: `ra`, `rb`, `rc`, `rd`, `rs`, `rx`, `ry`, `rz` There is one additional general purpose register (`rr`), but it is reserved for use by the assembler.
In addition to the general purpose registers, there is also a stack pointer (`sp`) and a program counter (`pc`).
Alongside the registers, there are four status bits/flags which are automatically set by various instructions: `CF` (carry flag), `ZF` (zero flag), `OF` (overflow flag), and `SF` (sign flag).
In instructions, the registers are encoded via the following mappings to numbers:

- `ra` — 0
- `rb` — 1
- `rc` — 2
- `rd` — 3
- `rr` — 4
- `rs` — 5
- `rx` — 6
- `ry` — 7
- `rz` — 8
- `pc` — 9
- `sp` — 10

2.2 Address Space

The memory space is 28-bits and all values are stored as little-endian. Memory permissions and mappings are determined by the executable format.

Chapter 3

Instruction Reference

Note that all bit diagrams are in big-bit-endian order (i.e. the highest bit is first).

3.1 Type 1 Instructions

Type 1 instructions support two operands. While the first (`dst`) must always be a register, the second (`src`) can be either a register or a 28-bit constant value.
Instructions are encoded as follows:
When `src` is a...

- **register:** `010oooossssdddd000000`
- **constant:** `011ooooddddd000sssssssssssssssssssssssssss`

where `o` are the bits for the opcode, `s` are the bits for the `src`, and `d` are the bits for the `dst`.
Unless noted otherwise, the flags are set as follows:

- The `ZF` flag is set if the result is 0
- The `OF` flag is set if the signed result overflows the 28-bit limit
- The `CF` flag is set if the unsigned result overflows the 28-bit limit
- The `SF` flag is set if the signed result is less than 0

3.1.1 ADD

The `ADD` instruction adds the value of `src` to `dst` and stores it in `dst`.
Opcode: 0
Allowed src: Register, Constant
Allowed dst: Register
Flags Set: `ZF`, `OF`, `CF`, `SF`

3.1.2 SUB

The `SUB` instruction subtracts the value of `src` from `dst` and stores it in `dst`.
Opcode: 1
Allowed src: Register, Constant
Allowed dst: Register
Flags Set: `ZF`, `OF`, `CF`, `SF`

3.1.3 MUL

The `MUL` instruction calculates the signed multiplication of `src` and `dst` and stores it the high four by7es into `rd` and the low four bytes into `dst`. The low bytes take precedence. If you are confused about the semantics of this instruction with regards to signed values, I recommend checking the `imul` instruction from `x86`.
Opcode: 2
Allowed src: Register, Constant
Allowed dst: Register
Flags Set: `ZF`, `OF`, `CF`, `SF`

3.1.4 DIV

The `DIV` instruction calculates the signed division of `dst` by `src` and stores it in `dst` and stores the remainder into `rd`. The quotient takes precedence. If you are confused about the semantics of this instruction with regards to signed values, I recommend checking the `idiv` instruction from `x86`.
Opcode: 3
Allowed src: Register, Constant
Allowed dst: Register
Flags Set: `ZF`, `OF`, `CF`, `SF`

3.1.5 CMP

The `CMP` instruction acts as the `SUB` instruction, except that the result is not actually stored in `dst`.
Opcode: 4
Allowed src: Register, Constant
Allowed dst: Register
Flags Set: `ZF`, `OF`, `CF`, `SF`

3.1.6 TST

The `TST` instruction acts as the `AND` instruction, except that the result is not actually stored in `dst`.
Opcode: 5
Allowed src: Register, Constant

Allowed dst: Register
Flags Set: `ZF`, `OF`, `CF`, `SF`

3.1.7 AND

The `AND` instruction calculates the bitwise-and of `src` and `dst` and stores it in `dst`.
Opcode: 6
Allowed src: Register, Constant
Allowed dst: Register
Flags Set: `ZF`, `OF`, `CF`, `SF`

3.1.8 ORR

The `ORR` instruction calculates the bitwise-or of `src` and `dst` and stores it in `dst`.
Opcode: 7
Allowed src: Register, Constant
Allowed dst: Register
Flags Set: `ZF`, `OF`, `CF`, `SF`

3.1.9 XOR

The `XOR` instruction calculates the bitwise-xor of `src` and `dst` and stores it in `dst`.
Opcode: 8
Allowed src: Register, Constant
Allowed dst: Register
Flags Set: `ZF`, `OF`, `CF`, `SF`

3.1.10 STR

The `STR` instruction stores the value in `dst` into memory at the address `src`.
Opcode: 9
Allowed src: Register, Constant
Allowed dst: Register
Flags Set: None

3.1.11 STB

The `STB` instruction stores the value of the low by7e of the value in `dst` into memory at the address `src`.
Opcode: 10
Allowed src: Register, Constant
Allowed dst: Register
Flags Set: None

3.1.12 LOD

The `LOD` instruction loads the value at the address `src` in memory into `dst`.
Opcode: 11
Allowed src: Register, Constant
Allowed dst: Register
Flags Set: `ZF`, `OF`, `CF`, `SF`

3.2 Type 2 Instructions

Type 2 instructions support only one operand (`val`), which for most instructions can be either a register or a 28-bit constant.
When `val` is a...

- **register:** `100ooooovvvv000`
- **constant:** `101ooooovvvvvvvvvvvvvvvvvvvvvvvvvv`

where `o` are the bits for the opcode and `v` are the bits for `val`.

3.2.1 JMP

The `JMP` instruction performs a relative jump forward by `val`.
Opcode: 0
Allowed val: Register, Constant
Flags Set: None

3.2.2 JLT

The `JLT` instruction performs a relative jump forward by `val` if `SF` $\neq$ `OF`.
Opcode: 1
Allowed val: Register, Constant
Flags Set: None

3.2.3 JEQ

The `JEQ` instruction performs a relative jump forward by `val` if `ZF` $= 1$
Opcode: 2
Allowed val: Register, Constant
Flags Set: None

3.2.4 CAL

The `CAL` instruction performs a relative call forward by `val`. It does so by first pushing the current value of `pc` plus the size of the current instruction onto the stack (by decrementing

`sp` by 4 and then storing the current value of `pc` into where `sp` used to point) and then perform a relative jump foward by `val`.
Opcode: 3
Allowed val: Register, Constant
Flags Set: None

3.2.5 PSH

The `PSH` instruction stores `val` into the address in `sp`, and then decrements `sp` by 4.
Opcode: 4
Allowed val: Register, Constant
Flags Set: None

3.2.6 POP

The `POP` instruction loads the value from `sp` + 4, and then increments `sp` by 4.
Opcode: 5
Allowed val: Register
Flags Set: `ZF`, `SF`

3.2.7 NOT

The `NOT` instruction performs a bitwise-not of the value in `val`.
Opcode: 6
Allowed val: Register
Flags Set: `ZF`, `SF`

3.2.8 OUT

The `OUT` instruction prints the low by7e of `val` to stdout.
Opcode: 7
Allowed val: Register, Constant
Flags Set: None

3.2.9 INP

The `INP` instruction reads a character from stdin and stores it into the by7e at the address `val`.
Opcode: 8
Allowed val: Register, Constant
Flags Set: None

3.2.10 AMP

The `AMP` instruction is the absolute analogue of `JMP`.
Opcode: 9
Allowed val: Register, Constant
Flags Set: None

3.2.11 ALT

The `ALT` instruction is the absolute analogue of `JLT`.
Opcode: 10
Allowed val: Register, Constant
Flags Set: None

3.2.12 AEQ

The `AEQ` instruction is the absolute analogue of `JEQ`.
Opcode: 11
Allowed val: Register, Constant
Flags Set: None

3.2.13 AAL

The `AAL` instruction is the absolute analogue of `CAL`.
Opcode: 12
Allowed val: Register, Constant
Flags Set: None

3.3 Type 3 Instructions

Type 3 instructions do not take any operands.

3.3.1 RET

The `RET` instruction pops the top value off of the stack into `PC`.
Instructions are encoded as follows: `11o0000` where `o` is the bit for the opcode.
Opcode: 0
Flags Set: None

3.3.2 NOP

The `NOP` instruction ("no operation") does nothing.
Opcode: 1
Flags Set: None

3.4 Changes

3.4.1 Version 1.5

- Clarify the semantics of `DIV` and `MUL` by referencing analagous instructions from `x86`.

3.4.2 Version 1.4

- Specified what happens if `DIV` or `MUL` act on `RD`.

3.4.3 Version 1.3

- Added register number for `SP`.