Toxic Processor

A simplistic 4-bit processor ready to synthesis

Version 2.0.0

Entropy Xu
entropy.xcy@protonmail.com

May 9, 2019

Contents

Chapter 1

Introduction

1.1 Intention

1.2 Advantages

1.3 Limitations

1.4 History of Revisions

Chapter 2

Design of the Processor

2.1 Overview

This processor is a **4-bit Stack machine** (0-address machine).

- The addressing width is configurable to be a multiple of 4.
- The stack depth is configurable to be greater than 16.
- The width of each Instruction is 4 bits. Thus, 16 Instructions in total.
- The width of each block inside the stack is 4 bits.
- Von Neumann Architecture: Code memory and data memory are using the same addressing space.

2.2 Data Structure

The name **stack machine** or equivalently 0-address machine means that there is no addressable register in this machine neither do operands in the instructions.

In order to store temporary data in this processor, we use a hardware stack to replace the register file which is usually implemented by other popular processors.

The 4-bit block-data-width will not limit the scalability of this processor in that addressing width of this processor is 4-bits but a configurable width of a multiple of 4 (usually 8 bits or 12 bits or 16 bits).

2.2.1 Stack for Storing Temporary Data

We have a stack for storing temporary data. Stack is a LIFO (Last In First Out) data structure. Each block of the stack is a 4-bits register. The stack supports common operations like push and pop. For each Instruction we execute, we will have to read value from TOS (Top of Stack) and NTOS (Next Top of Stack), and push the result of the operations to the stack. (Refer to Figure 2.1 for the model of the stack)

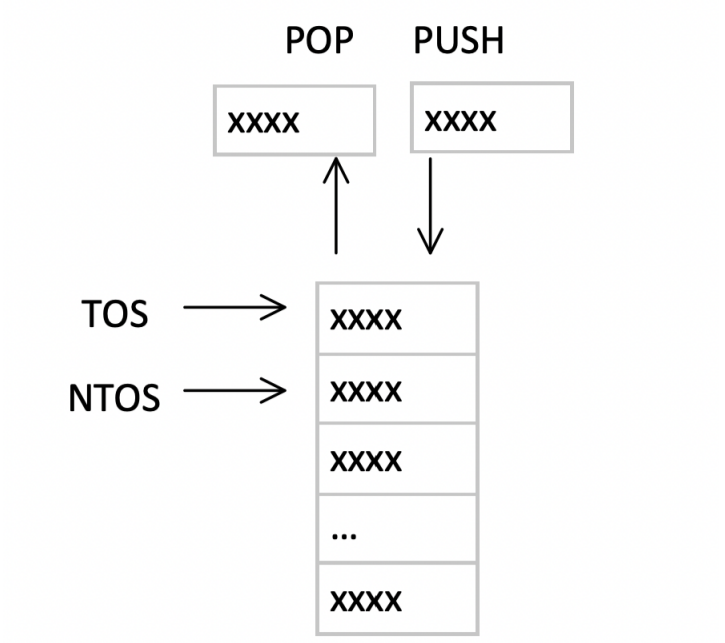

Figure 2.1: Toxic Stack Model

2.2.2 Queue for Addressing Bus

We have discussed that the addressing width for the Toxic processor is configurable and usually more than 4 bits. Thus, we establish a Queue as the data structure for storing the address of the Bus. For the length of the queue, we have $Length(Queue) = BitWidth(BusAddress)/4 * 2$

Queue is a FIFO data structure. However, the Queue data structure we used in the Toxic processor is similar to the common Queue but with some tweaks. Same as the stack, each block in the Queue is a 4-bits register. The

Queue should supports common operations like **enqueue** and **dequeue**. Refer to Figure 2.2 for the model of the queue and the connection with the BusAddress. Noted that, when we **enqueue**, all the blocks shift left one position relative to the BusAddress; when we **dequeue**, all the blocks shift right one position relative to the BusAddress.

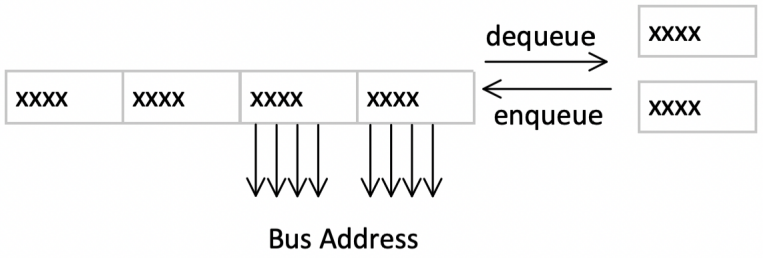

Figure 2.2: Toxic Queue Model

2.2.3 Memory

The memory of the Toxic processor is accessed through Bus and only takes part of the addressing space. The model of the Memory is different from common RAMs since for common RAMs, the block bit width is 1 bype (8 bits) while for the Toxic processor, the width of a block is half-byte (4 bits).

For the Toxic processor, we assign the width of each block of the memory to be 4 bits. This memory model can be easily implemented using a standard memory block and would be discussed in detail in Chapter 4. Detailed addressing space definitions can be found in detail in section 2.4.

2.2.4 Carry Bit

Carry Bit is a single bit register storing the carry of operations. There are three instructions that may possibly generate a carry: ADD, LS, RS. Please refer to subsection 3.4.1 for ADD, 3.4.3 for LS, 3.4.4 for RS. In addition, the instruction CMP allows programmers to access the carry bit. Please Refer to 3.7.1 for details.

2.3 Addressing Modes

There are three addressing modes in the Toxic processor:

- TOS (Top of Stack)
- NTOS (Next Top of Stack)
- Bus for memory and peripherals

2.4 Addressing Space

These definitions for the addressing space is for the standard version of the Toxic processor. More versions of definitions of implementations can be found in Chapter 4

2.4.1 Reserved: 0x0-0xf

2.4.2 Code Memory: 0x10-0x7fff

2.4.3 Data Memory: 0x8000-0xefff

2.4.4 Peripherals: 0xf000-0xffff

Chapter 3

Instruction Set Architecture

3.1 Instructions Map

1:0\3:2	00	01	11	10
00	P1	POP	ADD	SV
01	P11	DIS	NAND	LD
11	CMP	SWP	LS	B1
10	PC	RVS	RS	B0

For the explanation of instructions below, refer to Chapter 2 for details about terminologies like Stack, Queue, pop, push, enqueue, dequeue, TOS, NTOS.

3.2 Push Instructions

3.2.1 P1

- Functionality: **Push** 0001 to stack.
- Expression:

```
Stack.push(0001);
```

3.2.2 P11

- Functionality: **Push** 0011 to stack.

- Expression:

```
Stack.push(0011);
```

3.3 Stack and Bus Operation Instructions

3.3.1 POP

- Functionality: **Pop** from stack and **enqueue** the value **poped** from stack to the queue.
- Expression:

```
val = Stack.pop();
Queue.enqueue(val);
```

3.3.2 DIS

- Functionality: **Pop** from the stack and discard the value.
- Expression:

```
Stack.pop();
```

3.3.3 SWP

- Functionality: Swap the value of **TOS** and **NTOS**.
- Expression:

```
val = Stack.ntos;
Stack.ntos = Stack.tos;
Stack.tos = val;
```

3.3.4 RVS

- Functionality: **dequeue** from the Queue and **push** the value **dequeued** to the Stack.
- Expression:

```
val = Queue.dequeue();
Stack.push(val);
```

3.4 Numeric Computing Instructions

3.4.1 ADD

- Functionality: **Push** the value of **TOS** + **NTOS**.
- Expression:

```
val = Stack.tos + Stack.ntos;
Stack.push(val);
```

- Carry bit: 1 for addition having an carry, 0 for addition without an carry.

3.4.2 NAND

- Functionality: **Push** the value of **TOS** NAND (bit-wise) **NTOS**.
- Expression:

```
val = ~(Stack.tos & Stack.ntos);
Stack.push(val);
```

3.4.3 LS

- Functionality: **Left Shift** one bit the value of **TOS**.
- Expression:

```
Stack.tos = Stack.tos << 1;
```

- Carry bit: carry bit equals the most significant bit of **TOS before shifting.**

3.4.4 RS

- Functionality: **Right Shift** one bit the value of **TOS**.
- Expression:

```
Stack.tos = Stack.tos >> 1;
```

- Carry bit: carry bit equals the least significant bit of **TOS before shifting.**

3.5 Memory Operations Instructions

3.5.1 SV

- Functionality: **Save** the value of **TOS** to memory location pointed by the **Bus** address.
- Expression:

```
Memory[BusAddress] = Stack.tos;
```

3.5.2 LD

- Functionality: **Push** the value of at memory location pointed by the **Bus** address to the **Stack**.
- Expression:

```
Stack.push(Memory[BusAddress]);
```

3.6 Branch Instructions

3.6.1 B1

- Functionality: Branch to the address pointed by the **Bus** address if the least significant bit of **TOS** is 1.
- Expression:

```
if(Stack.tos[0] == 1)
{
    PC = BusAddress;
}
```

3.6.2 B0

- Functionality: Branch to the address pointed by the **Bus** address if the least significant bit of **TOS** is 0.
- Expression:

```
if(Stack.tos[0] == 0)
{
    PC = BusAddress;
}
```

3.7 Special Instructions

3.7.1 CMP

- Functionality: Compare **TOS** and **NTOS** with the assumption that they are both signed values, push an output (described below) to the Stack.
- Output[0] equals 1 for **TOS** == **NTOS**, equals 0 otherwise.
- Output[1] equals 1 for **TOS** ¿ **NTOS**, equals 0 otherwise.
- Output[2] equals 1 for **TOS** ¡ **NTOS**, equals 0 otherwise.

- Output[3] equals 1 for Numeric operation having an carry, equals 0 otherwise. ADD, LS, RS are the three instructions that can possibly generate a carry. Please refer to subsection 3.4.1 for ADD, 3.4.3 for LS, 3.4.4 for RS.
- Expression:

```
output0 = Stack.tos == Stack.ntos;
output1 = Stack.tos > Stack.ntos;
output2 = Stack.tos < Stack.ntos;
output3 = carry;
Output = output0 + output1 << 1
        + output2 << 2 + output3 << 3;
Stack.push(Output);
```

3.7.2 PC

- Important Note: This Instruction is optional for the most simplistic design.
- Functionality: Replace the whole **BusAddress** with current **PC**.
- Recall that **BusAddress** is part of the **Queue**. Please refer to 2.2.2 for more details about BusAddress and Queue.
- Expression:

```
BusAddress = PC;
```

Chapter 4

Micro-Architecture

4.1 Modules List

4.2 Data Path

Chapter 5

Assembly Language

In this chapter we will talk about how to write assembly code for the Toxic processor in a programmer's perspective. Useful Examples will also be provided.

5.1 Format

- Each Instruction should take one line.
- Put comments after ";".
- Use "," to connect several instructions in one line.
- Use ":" to tag an location in code memory.

5.2 Useful Examples

5.2.1 Push 0000 to Stack

Number of Instructions: 2

```
P1 ; TOS now is 0001
RS ; 0001 >> 1 = 0000
```

5.2.2 Push 1110 to Stack

Number of Instructions: 10

```
P11; TOS now is 0011
LS,LS; TOS now is 1100
P1,LS; TOS now is 0010, NTOS now is 1100
ADD; Stack is: 1110, 0010, 0011
SWP,DIS,SWP,DIS; discard 0010 and 0011
;now TOS is 1110
```

5.2.3 Push current TOS to Stack (Duplicate TOS)

Number of Instructions: 5

```
P1, RS; Push 0000 to Stack
ADD; TOS + 0000 = TOS
SWP, DIS; discard 0000
```

5.2.4 Perform AND operation (Bitwise)

Number of Instructions: 9

$$XorY = (XnandY)nand(XnandY)$$

```
; Suppose TOS is 1010 and NTOS is 1100
NAND ; TOS = ~(1010 & 1100) = 0111
P1, RS, ADD, SWP, DIS; duplicate TOS
NAND ; TOS = ~(0111 & 0111) = 1000
SWP, DIS; discard 0111 we duplicated
```

5.2.5 Perform OR operation (Bitwise)

Number of Instructions: 25

$$XorY = (XnandX)nand(YnandY)$$

```
; Suppose TOS is 1010 and NTOS is 1100
P1, RS, ADD, SWP, DIS; duplicate TOS
NAND; TOS = 1010 nand 1010 = 0101
POP; reserve the current TOS in Queue
DIS, SWP; discard duplicated 1010
;then put 1100 on Top
```

```
P1, RS, ADD, SWP, DIS; duplicate 1100
NAND; TOS = 1100 nand 1100 =  0011
POP; reserve the current TOS in Queue
DIS, SWP; make the original Stack Intact
RVS, RVS; restore 0101 and 0011 from Queue
NAND; perform nand operation
; now, TOS = 1010 or 1100 = 1110
SWP, DIS, SWP, DIS; discard 0101 and 0011
```

5.3 Pseudo Codes