A Behavioral-level Verilog Implementation of the TERA Instruction Set Architecture

Brian Hong, Luka Lipovac, &
Arthur Christopher Watkins

Department of Electrical Engineering

The Cooper Union for the Advancement of Science & Art
ECE-151: Computer Architecture
Prof. Rob Marano, Instructor

May 14, 2016

Contents

List of Figures

List of Tables

1 Introduction

The **E**ight-bit **R**egister **A**rchitecture Instruction Set Architecture (TERA ISA) is an 8-bit, reduced instruction set architecture that employs a total of sixteen instructions, each paired with a specialized function register. The TERA ISA is unique in this, that each instruction is paired with a specific register, and allows for a greater amount of inherent instructions and registers than may normally be possible for a standard 8-bit architecture. Although this novel idea may seem difficult to grasp at first, it quickly becomes intuitive as one familiarizes himself with the intricacies of the design. Ultimately, the TERA ISA is extremely versatile and permits enormous flexibility with regards to software development in an 8-bit environment.

A behavioral-level Verilog implementation of the TERA ISA was created to allow for the computer to be tested and modified with ease. To accompany the Verilog implementation of the TERA ISA, an assembler with error checking was created for translating TERA assembly language to machine language, providing the developer with the ability to easily test TERA programs. Both leaf-procedure and nested-procedure test programs were created to demonstrate the functionality of the TERA assembly language and the TERA ISA.

2 Description of Instruction Set Architecture

The TERA ISA is classified as a Harvard architecture, which means that memory used in this implementation is separated into two categories: instruction memory, and data memory. At the initialization of a program, a set of machine level instructions are loaded into the instruction memory of the computer, with the first instruction in the program corresponding to the first position in memory. The instruction memory can only be read and executed, while the data memory can only be read or written to. (This is analogous to the *.text* section and the *.data* section of a program.)

The TERA Computer is also a single-cycle processor, which means that every instruction only takes one clock cycle to execute, from the fetch step all the way to the end of the execute step.

The instructions can be in either one of two formats: R-format or I-format. As mentioned above, most of the instructions have the R-format of two 4-bits that refer to a register.

Basic Instruction Formats		
Type	**Higher Nibble (f)**	**Lower Nibble (s)**
I	[instruction]	[constant]
R	[instruction] = R[f]	[register] = R[s]

Table 1: Basic instruction formats (8-bit instructions)

This may sound confusing at first, but it's a way to increase the number of instructions while keeping the number of registers the same. A typical architecture might have the following instruction format structure: *[opcode] [r1] [r2]*.

The TERA architecture is as if all the registers were mapped to each opcode, which would give *[opcode] [r1/opcode] [r2]*, where the r1 is the same as the opcode. This then can be reduced to *[r1/opcode] [r2]*, which TERA uses.

Register List		
Name	**Number**	**Purpose**
$zero	0	Holds constant value 0
$arg0 ($not)	1	General purpose register
$and	2	Used by **and** instruction
$or	3	Used by **or** instruction
$add	4	Used by **add** instruction
$rlf	5	Used by **rlf** instruction
$rrt	6	Used by **rrt** instruction
$sle	7	Used by **sle** instruction
$sge	8	Used by **sge** instruction
$arg1 ($bfs)	9	General purpose register
$jal	10	Used by **jal** instruction
$arg2 ($lli)	11	General purpose register
$arg3 ($lhi)	12	General purpose register
$lw	13	Used by **lw** instruction
$sw	14	Used by **sw** instruction
$mov	15	Used by **mtr** and **rtm** instructions; ***NON**-addressable!*

Table 2: Complete list of registers

This, each register being mapped to one specific instruction, means that the instructions will have to utilize that register, making them special-purpose registers. They can be considered a 'pseudo-acculumator' for each instruction. There are a few exceptions; registers like **$not**, which corresponds to a NOT operation, can be used as a general purpose register. This is possible because the NOT operation is unary so it operates on only one register, unlike instructions like AND and OR which, as binary operators, necessitate two registers. While TERA was being designed, the second register was designated for storing results to allow for these general purpose registers to exist.

The **mtr** and **rtm** instructions are special in that they will execute the

'**move**' operation even if it is used as the second register. The **move** instruction/register is the only way to move data around in the TERA architecture. When **move**'s opcode (1111) is in the first 4 bits of the instruction, it moves the data from the move register into the specified register. If **move**'s opcode (1111) in the second 4 bits, data from the specified register is moved into the **$mov** register.

There are two I format instructions: load-lower-immediate and load-higher-immediate. For these two instructions, the **$mov** register acts as the 'accumulator'; it will load either the higher or lower 4 bits with the supplied immediate 4 bits. This frees up the registers themselves to be used as general purpose registers as well.

See section 4 "Sample Code" for a few example code snippets and for better understanding of how the code works. Continue in that section to better understand exactly what each instruction does, and for an illustration of the data path.

Core Instruction Set			
Name	**Mnemonic**	**Format**	**Operation**
Not	not	R	R[s] = ~R[s]
And	and	R	R[s] = R[f] & R[s]
Or	or	R	R[s] = R[f] | R[s]
Add	add	R	R[s] = R[f] + R[s]
Rotate Left	rlf	R	R[s] = rotl(R[f])
Rotate Right	rrt	R	R[s] = rotr(R[f])
Set Less Than Equal	sle	R	CF = R[f] $\leq$ R[s]
Set Greater Than Equal	sge	R	CF = R[f] $\geq$ R[s]
Branch If Flag Set	bfs	R	CF ? PC = R[s] : ++PC
Jump And Link	jal	R	R[s] = PC + 1; PC = R[f]
Load Lower Imm.	lli	I	R[15] = {4'b0, s}
Load Higher Imm.	lhi	I	R[15] = {s, 4'b0}
Load Word	lw	R	R[s] = M[R[f]]
Store Word	sw	R	M[R[f]] = R[s]
Move To Register	mtr	R	R[s] = R[f]
Register To Move	rtm	R	R[f] = R[s]

Table 3: Core instruction set

not: R format instruction that performs a bit-wise NOT operation on R[s] (and stores the result into R[s])

and: R format instruction that performs a bit-wise AND operation with the values in R[f] and R[s] and stores the result in R[s]

or: R format instruction that performs a bit-wise OR operation with the values in R[f] and R[s] and stores the result in R[s]

add: R format instruction that performs a binary addition of values in R[f] and R[s] and stores the result in R[s]. Carry bit is discarded.

rlf: R format instruction that performs a binary rotate-left operation to R[f] and stores the result in R[s]

rrt: R format instruction that performs a binary rotate-right operation to R[f] and stores the result in R[s]

sle: R format instruction that performs a comparison between R[f] and R[s] and sets Comparison Flag high if R[f] $\leq$ R[s]

sge: R format instruction that performs a comparison between R[f] and R[s] and sets Comparison Flag high if R[f] $\geq$ R[s]

bfs: R format instruction that sets the value of PC to the value stored in R[s] if the Comparison Flag is set to high

jal: R format instruction that sets the value in R[s] to the current value of PC+1 and then sets the value of PC to the value stored in R[f]

lli: I format instruction that sets the value of the most significant 4 bits of **$mov** with the indicated immediate (a possible range of values from 0-14)

lhi: I format instruction that sets the value of the least significant 4 bits of **$mov** with the indicated immediate (a possible range of values from 0-14)

lw: R format instruction that loads R[s] from memory address held in R[f]

sw: R format instruction that stores R[s] to memory address held in R[f]

mtr: R format instruction that sets the value of the R[s] register to the value that is currently stored in **$mov**

rtm: R format instruction that sets the value of **$mov** to the value that is currently stored in R[s]

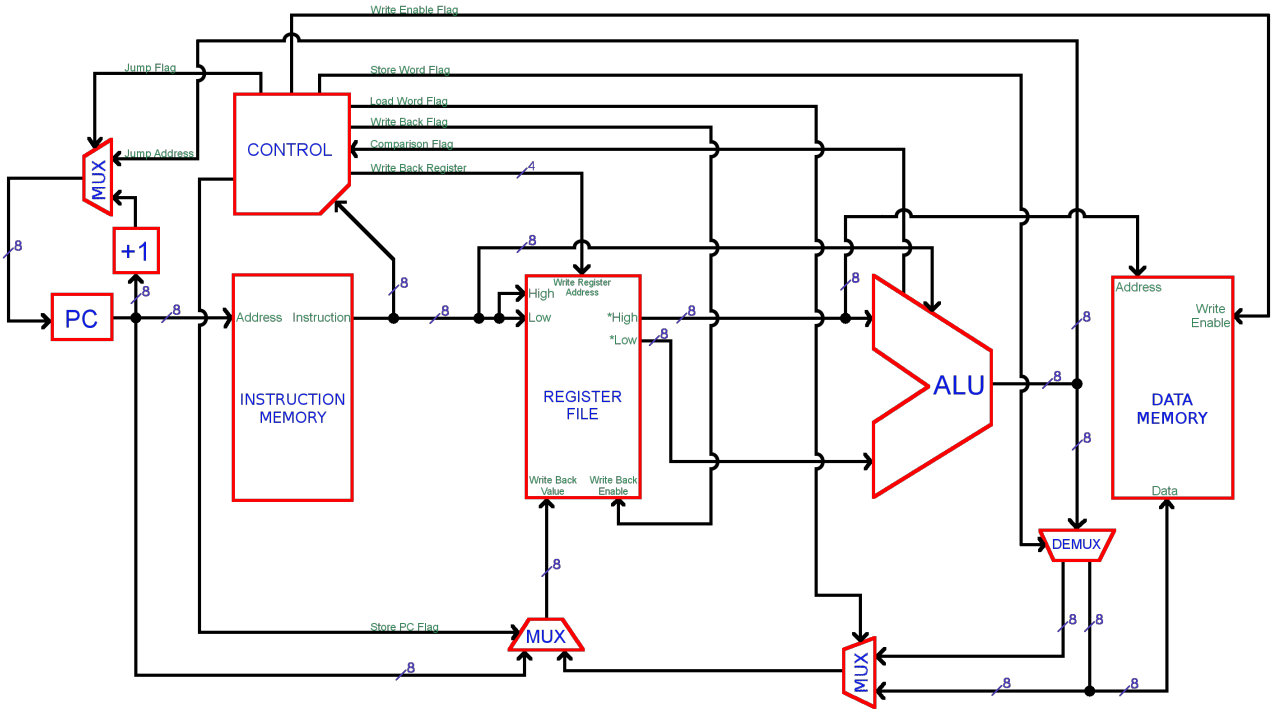

Figure 1: Outline of data path

3 Explanation of Design Choices

The most important and obvious choice that was made with regards to the TERA ISA was the unconventional combination of instructions and registers. Due to the very small number of bits allotted for addressing either registers or instructions, only a range of combinations from 16 instructions and 4 registers to 4 instructions and 16 registers can be achieved. This was viewed as a major problem. With the implementation of an ISA wherein the instructions correspond to registers, TERA was able to achieve an unprecedented 16 instructions and 16 registers. For an 8-bit architecture, this design allows for great versatility when writing programs, but while it is very powerful at this scale, it is not very useful for architectures larger than 8-bits, which was a significant compromise that was made while designing the TERA ISA.

For TERA to be implemented as it is, the TERA ISA had to stray far from a more standard ISA. This is evidenced by the presence of the move register, corresponding to the instruction 1111, in TERA. Due to many of the instructions in TERA executing the instruction on both their own register and the second listed register and then storing them into the second, values from registers need to be moved so as to allow for proper computation. For example: if two numbers were added – the one present in **$add** and the one present in **$or** – then the values present in **$rrt** and **$sle** need to be OR'ed, the value present in **$or** would need to be moved to another register and either the **$rrt** or **$sle** value would need to be copied into **$or**, and then the **or** instruction would need to be executed. Acting as a "pseudo-accumulator", values that need to be moved from register to register are first stored into **$mov** and then are copied from **$mov** to the desired destination register. **$mov** is also the location in which immediate values are stored when either **lli** or **lhi** are called. Due to how **$mov** works and its location at 1111, there is a restriction in regards to the **lli** and **lhi** instructions: only values from zero through fourteen (0000 through 1110) can be loaded as immediate values. A value of 255 (11111111) can be achieved by loading from **$zero** and using the **not** instruction.

Another way in which TERA strays away from a standard architecture is the inclusion of a rotate left and right instead of a shift left and right. This choice was made because while shift is a very useful instruction, rotate is significantly more versatile, and it may be used as a shift by bit-masking.

Some more unconventional design choices were made with TERA, including the exclusion of subtraction, multiplication, and division instructions.

Although TERA does have room for a very large amount of instructions, it was determined that these complex arithmetic instructions were unnecessary in this RISC-based architecture. To do subtraction with TERA, the intended subtrahend can be written using the concept of two's complement, then simply added to the minuend to give the difference. For both multiplication and division, a simple iterative loop can be created that takes the multiplicand/-dividend and adds/subtracts from itself its value (with the method presented above) however many times is specified by the multiplier. By subtracting 1 from the multiplier/divisor and checking if it is equal to **$zero** on each pass of the loop, one can be confident that the correct calculation was carried out.

The inclusion of the **$zero** register into TERA was very useful for a variety of reasons. The register is set to store a constant 8-bits of 0 (00000000). The register having 8 bits constantly set to zero simplifies loops in our ISA in a significant number of cases–two of which being loops used in multiplication and division as previously mentioned–since an immediate of zero does not have to be loaded into a register for comparisons. The value stored in the **$zero** register can be loaded into other registers to effectively clear them.

The final major decision that was made in regards of the design of TERA deals with what register values are stored in after an operation. predominantly done due to the difficulty that would be present with inverting a value with the **not** instruction if the opposite was done, it was made so that when a value in TERA is calculated the result is usually stored in the second specified register. This decision to store values in the second register specified by every instruction also has allowed for TERA to have four general purpose registers: $arg0 (0001), $arg1 (1001), $arg2 (1011) and $arg3 (1100). These registers correspond with instructions that do calculations on only one register. In TERA's case, these registers are **$not**, **$bfs**, **$lli**, and **$lhi**. If any of these four registers (or their aliases, **$arg0** through **$arg3**) are specified as the second register, the value of a calculation is stored into them. Due to the nature of the general purpose registers' corresponding instructions, **not**, **bfs**, **lli**, and **lhi** can be performed without the value present in the corresponding registers affecting the operation nor being modified.
Another compromise made was the 'jal' instruction. It would load the return address to R[s] and jump to R[f], not the other way around, so that if there was a need to jump without linking (which would take up another register by overwrite), *jal $zero* can be called. When any data is written to **$zero**, the data just get deleted, like a */dev/null* on a Unix machine.

4 Sample Code

This section provides a set of sample code snippets to demonstrate and give ideas of how some code behavior can be achieved using the TERA assembly.

```
lli 0110
mtr $not
add $not
```

To get started and get familiar with the TERA asm, the above code snippet will load the lower 4 bits of the move register with 0110, move that number **$not** register, and then add it to the value in **$add**. Any subsequent *add $not* will increment the value in it by 6, as long as the value in $not stays constant.

```
rtm $zero # get zero (00000000)
mtr $bfs  # move zero to $bfs
not $bfs  # becomes 11111111
add $bfs  # add -1 (subtract 1)
```

The TERA instruction set does not have a subtract instruction. The instruction was thought redundant as subtraction can be achieved by adding a negative number. The above code demonstrates subtraction. It gets the value 0 from the $zero register, and then nots it, which results in 11111111. It can be interpreted as 255, but when added to a non-zero number, it substracts one. This works because of two's complement method of adding and discarding the carry bit.

```
 lli 0101
 mtr $not # lower 4 bits are 0101
 lhi 0001
 mtr $or  # higher 4 bits are 0001
 or $not  # not becomes 0001 0101
 rtm $not # load value from $not
 mtr $jal # and put it in $jal
jal $zero # jump to 0001 0101
```

The first 5 lines of the above snippet demonstrates loading 8 bit wide immediate values. The next two lines show how data from one register can be moved to another using the move register and the mtr/rtm psuedo-instructions. The last line jumps to the address loaded from the first part of the code. Notice it tries to store the return address to $zero, which accepts it, but throws the data away.

5 How to Use the Toolchain

An assembler was also written using C to make the building and testing of the programs easier. To get the source code:

```
$ git clone https://github.com/archwa/tera_computer.git
$ cd tera_computer
$ make
```

Running '**make**' in this new directory will build the computer, assembler, and the program. By default, it will assemble program.asm in the directory *./asm/program/*. In the same directory, there are 2 other sample programs provided: leaf.asm and nested.asm. The program leaf.asm shows how a leaf procedure in TERA assembly looks like and demonstrates the computer's ability to run such procedures by executing a routine that doubles a number until it is greater than 48. The program nested.asm demonstrates the computer's ability to perform nested procedures by recursively calculating a summation of numbers less than or equal to the number in $arg1. To assemble other programs besides *./asm/program/program.asm*, the input assembly filepath must be passed in through setting the program variable in the Makefile. This can be achieved by adding 'program=[path/to/assembly/program]' after make, like so:

```
$ make program=asm/program/nested.asm # and then
$ make run
```

There are also phony targets '*run*' and '*clean*'. '**make run**' will run the compiled Verilog computer, and '**make clean**' will remove all built files. If, for any reason, the computer hangs during its program execution,^c or [ctrl-c] will halt it. If there is a single greater than sign '>' present in the terminal, enter the text 'finish' to exit out of the Verilog interpreter.