Technical Reference for the *dt* Programming Language and Assembler

Justin Severeid and Elliott Forbes
Department of Computer Science
University of Wisconsin-La Crosse
La Crosse, WI 54601
{severeid.justin, eforbes}@uwlax.edu

Technical Report TR09122024
University of Wisconsin-La Crosse

1. Introduction

This document is a technical reference for the DuctTape (*dt*) programming language, and its compiler/assembler (also called `dt`). Rationale, and the background for this language/tool are described in [3] and will not be repeated here. If you have used *dt* and found it helpful in your research, please cite [3] rather than this technical report.

The most up-to-date version of document can always be found at https://cs.uwlax.edu/~eforbes/dt/dtref-recent.pdf. It will be updated as bugs are found/fixed and as new versions of *dt* are released. Older versions of this document will also be available at the same link, where the document name will follow the form `dtref-TRmmddyyyy.pdf`, where `mm` is the two digit month, `dd` is the two letter day, and `yyyy` is the four letter year of the release date of the document.

For the sake of clarity, this document will stylize DuctTape the language as *dt*, with italicized font. DuctTape the high-level assembler tool that implements the *dt* language will be referred to using a fixed-width system font as `dt`. And we will use `.dt`, a fixed-width system font prefixed with dot (i.e. referring to a dot-dt file), to refer to a program written in the *dt* language.

The remainder of this document outlines the installation and usage of the *dt* high-level assembler in Section 2. The details of the *dt* syntax are discussed in Section 3. Section 3 also describes what code will be emitted by `dt` for high-level language constructs. The output file formats of `dt` will be explained in detail in Section 4. A brief walk-through of the `dt` high-level assembler itself can be found in Section 5. And finally, knowns bugs and limitations of `dt` are enumerated in Section 6.

2. Installation and Command-line Options

This section helps you get started with using `dt`, the high-level assembler for the *dt* language.

2.1. Installation

The source code for the `dt` high-level assembler has been moved to a public GitHub repository, which can be found at https://github.com/eforbes-uwl/dt. Use the usual `git` commands to clone the repository:

```
git clone https://github.com/eforbes-uwl/dt.git
```

Once cloned, you can descend into the `./dt/` base directory, and compile. Simply issue the `make` command to compile, there is only one compilation target defined to build `dt`, specifically `./dt/bin/dt`. The `clean` build target is also defined to delete all derived files, including the `dt` executable itself. The only requirements for compilation are reasonably recent versions of `gcc`, `flex` and `bison`. `dt` was most recently compiled with `gcc` 14.2.1, `flex` 2.6.4, and `bison` 3.8.2, although there is no reason to believe older/newer versions of these tools won't compile.

```
cd ./dt
make
```

You may want to add the `dt` executable to your `$PATH` environment variable. The `dt` executable will be in the `./dt/bin` subdirectory. Edit your `.bashrc` (or similar) file in your home directory, and add the `dt` path to the `PATH` environment variable, before it has been exported.

```
PATH="$HOME/.local/bin:$HOME/bin:$PATH:$HOME/dt/bin"
export PATH
```

2.2. Command-line Usage

The following help message is displayed when you execute `dt` from the command-line, without specifying any options or files.

```
usage: ./bin/dt [flags] <infile...>

        -version          Print the dt version number and exit.
        -out <outfile>    Output filename will have a base name of <outfile>.
                          The default is "a" if -out is not used.
        -checking         Prints debug info (encodings, addresses, etc)
                          for parsed program to stdout.
        -elf              Outputs an ELF64 Linux executable. The output
                          filename will use a .out extension.
        -text             Outputs file to a flat memory image, as an
                          ASCII encoded text file. The file extension will be
                          .txt.
        -bin              Outputs file to a flat memory image, as a
                          binary file. The file name will end with a .bin
                          extension.
```

This is a helpful reminder of the available options and required files needed when running `dt`. The angle-brackets indicate required tokens, and square-brackets indicate optional tokens.

The `-version` command-line flag will always print the `dt` version number, and immediately quit, regardless of the other command-line flags. This technical report describes `dt` version 0.2.0.

You must have at least one input file indicated, that should be an ASCII text file with correct *dt* source code. It is customary that the file name end with the `.dt` extension, but it is not required to have any particular file name. You can have a single *dt* program that extends over several `.dt` source files, they will be parsed in the order listed at the command-line.

By default output files will have the base name `a` regardless of the output file format. This default can be changed by passing the `-out` option, along with a new base file name. The file extension after the base name will be determined by which output format is selected.

It is not required to use any other command-line flags, however, by default `dt` will not emit any output – not to the console, nor to any files. You can simply pass an input file name and `dt` will check the syntax, but will otherwise produce nothing else.

To have `dt` produce output, you must use *any* combination of the remaining command-line flags: `-checking`, `-elf`, `-text`, and/or `-bin`. The `-checking` writes helpful information about the input file to `stdout`. All other command-line flags will produce an output file. If multiple output formats are specified, then multiple files will be produced. The `-elf` flag indicates a ELF64 executable, suitable for execution in Linux. The `-text` and `-bin` options will produce flat memory images, typically used for simulators. These file formats are documented in Section 4.

3. *dt* Language Syntax

This section describes the *dt* language in detail. The section is divided into discussions of the high-level organization of a *dt* program, the syntax of the language, and the code emitted by high-level language constructs. There are also several code examples to highlight the flexibility of *dt*.

3.1. *dt* Program Organization

Figure 1 shows the high-level organization of a *dt* program. Each *dt* program is composed of one or more `mem()` blocks. The `mem()` blocks are used to encompass zero or more instructions, data values, and/or high-level language constructs. A `mem()` block must be supplied with a starting address, which are assumed to be 64 bit. Multiple memory regions can be populated without the need to specify what values appear in the "gaps" between memory regions. Each instruction will use 4 bytes of memory, in program order, starting from the `mem()` block starting address. Each data value put in the program source will use bytes of memory determined by which assembler directive is used to provide the value. `dt` will automatically ensure alignment, based on the size of data/instruction used, padding with zero values.

A single *dt* program can be split into multiple files. However, a `mem()` block must appear in its entirety in a single file. By convention, those files are named with an extension of `.dt`. There are no restrictions on the number of `mem()` blocks or their starting addresses for a program targeted for the flat memory image output formats (either text or binary). However, when using the ELF64 output file format, the program entry point must be address `0x000000400000`. Thus, there must be a `mem()` block with that same address, and it is expected that normally the first addresses used in that `mem()` block will be instructions.

Figure 1. Program organization of a complete *dt* program.

3.2. Assembly Syntax

The syntax of the *dt* language can be sub-divided into several categories: comments, `mem()` blocks, definitions, instructions, data values, high-level assignments, and high-level block statements. White-space and linefeeds are ignored by the *dt* high-level assembler. Also, and very importantly, *dt* is case-**sensitive** for definitions (section 3.2.3), but case-**insensitive** for all other constructs.

This subsection will cover the syntax required if the programmer uses only the assembly language syntax permitted by *dt*. Section 3.3 discusses the *additional* high-level language syntax that is permitted by *dt*.

3.2.1. Comments. Comments can appear anywhere in the source file. They start with a hash (#) and extend to the remainder of the source line. Multi-line comments are not implemented in *dt*. An example comment is shown in Listing 1.

```
# This is a comment.
```

Listing 1. Code example of a comment

3.2.2. `mem()` Blocks. A collection of *nearly* all other syntactical constructs other than comments must be contained within a `mem()` block. The exception to this rule is discussed in Section 3.3.1. A `mem()` block indicates to the *dt* high-level assembler where in the memory space the code contained within the `mem()` block will be put. That, in turn, impacts instruction and data addresses, which themselves impact instruction offsets.

A `mem()` block is specified by the keyword `mem`. After the `mem` keyword, the starting address should be specified within parenthesis. The starting address should be formatted the same way as 64 bit integer values as specified in section 3.2.4. After the parenthesized starting address, the instructions and data of a `mem()` block should be encompassed in curly braces. An example of a `mem()` block, with a starting address of `0x000000400000`, is shown in Listing 2.

```
mem (0x000000400000) {
    # instructions, data, etc., should go here -- customarily indented
}
```

Listing 2. Code example of a `mem()` block

3.2.3. Definitions. Definitions allow the programmer to name several of the other *dt* programming constructs. A definition is a label followed by a colon follow by one of those program constructs. A label can be any alpha-numeric string that starting with a letter. The label can contain underscores (_), but no other punctuation is allowed. Remember that definitions are the only part of *dt* that is case-sensitive, so `label` is the different than `Label` is the different as `LABEL`. There is no restriction on the length of the label.

There are two classes of definitions, those that name an associated memory location, and those that name a register. Registers are of one of two form: using the assembler names (for example `$zero`, `$ra`, `$sp`, etc.) or of the generic numbered form `$x0`-`$x31` for the integer registers. Once a register is given a defined name, the instructions that follow can then refer to the name in instead of the register number.

Memory locations can also be named, by simply prefacing an instruction and/or a value with a label, followed by a colon. A named memory location can be used by other instructions to assist in forming addresses, or in leaving offset calculations to the high-level assembler.

Note that all definitions are global. Therefore, it is possible to refer to registers and/or memory locations that span different `mem()` blocks – and even across multiple `.dt` files. Also note that registers must be named with a register definition *before* that label can be used by any instructions, but memory definitions/uses can come in any order. Registers can be renamed as often as needed, but memory locations can typically only be labeled once except for circumstances when a memory location is internally named by the high-level assembler (discussed in Section 5).

Listing 3 shows two example definitions, one for a register, and another to name a memory location that holds an `addi` instruction. Since the `addi` instruction is the first (and only) element that requires a memory location, `bar` is naming memory location `0x000000400000`.

```
mem (0x000000400000) {
    foo: $x5  # gives $x5 a handy name
    bar: addi foo, $x0, 0x7
}
```

Listing 3. Code example of a register and memory definition

3.2.4. Data and Immediate Values. Values are used in two main ways in *dt* programs. Some instructions require immediate values or offsets. Alternatively, it is possible to simply initialize a memory location with a known value. In either use case, immediate values are still bound by the number of bits in the instruction encoding of memory location.

Integer values can be specified in either decimal or hexadecimal, and use syntax similar to the C programming language. A decimal value is specified using an optional sign (+ or −), followed by the numerical value. A hexadecimal value is specified with a `0x` followed by the hexadecimal value.

Floating point values are also possible. They are also denoted in the same way as in the C programming language with an optional sign, a whole value, a decimal point, a fractional value, and an optional exponent which is denoted with the letter `e`, an optional sign, and the value of the exponent.

Memory can be initialized with a starting value. The syntax to fill a memory location is to use one of the assembler directives listed in Table 1, followed by listing the value. One of the directives (`.stringz`) additionally permits strings to be defined. *dt* strings also follow C-style syntax – the must be enclosed in double-quotes, within which characters and/or escaped characters can appear. *dt* strings are always terminated with a single byte with value zero. Unlike C, *dt* does not allow for a single character (i.e. single-quoted) data value.

As alluded toward in Table 1, the different directives are used to specify data with different data widths. The table lists the number of bytes, depending on which directive used. For `.stringz`, the number of bytes will equal the number of characters, plus the NULL terminating character. Escape sequences will be a single byte, as expected from C. The `.stringz` example from Table 1 will occupy 13 bytes of memory. The *dt* high-level assembler will automatically align all data to its data size (i.e. a `.word` will be aligned to 4 bytes, a `.half` to 2 bytes, etc.)

Directive	Size (in bytes)	Data Format	Example
`.byte`	1	Decimal or hex	`.byte 0xff`
`.half`	2	Decimal or hex	`.half -1`
`.word`	4	Decimal or hex	`.word 0xdeadbeef`
`.long`	8	Decimal or hex	`.long 0`
`.float`	4	Single precision IEEE 754	`.float 3.14159`
`.double`	8	Double precision IEEE 754	`.double 6.022e23`
`.stringz`	variable	ASCII encoded string	`.stringz "Hello world\n"`

Table 1. Assembler directives for inserting data values into memory in *dt* programs

Listing 4 shows several examples that use values. Line 2 shows an `addi` instruction, where the immediate has the value -1. Since this `addi` instruction appears first in the `mem()` block, it will occupy addresses `0x000000400000` through `0x000000400003`. Line 3 populates memory locations `0x000000400004` through `0x000000400008` with the ASCII values for the string `"foo\n"`. Line 4 indicates a 4 byte `.word`, but must be aligned. Therefore, `dt` will pad (with zeros) memory addresses `0x000000400009` through `0x00000040000b` and the value `0xdeadbeef` will occupy the next 4 bytes of memory. Finally, Line 5 will populate memory with the 64 bit IEEE 754 encoded value for 2.998×10^8 – an 8 byte quantity that happens to already be aligned at its address of `0x000000400010` without the need for any additional padding.

```
1 mem (0x000000400000) {
2     addi $x1, $x0, -1
3     .stringz "foo\n"
4     .word 0xdeadbeef
5     .double 2.998e8
6 }
```

Listing 4. Code example of immediate values and initialized memory locations

3.2.5. Instructions. Instructions can be specified by the instruction mnemonic, following the syntax in the RISC-V ISA specification [4] for the RV64I and RV64M subsets of the instruction set. Some instructions have alternative forms that will be discussed in section 3.3.1. Table 2 lists the valid mnemonics that can be used.

Listing 5 gives examples of several types of instructions. Instructions are fully specified using their mnemonic, followed by their operands. Operands can be specified in several ways, depending on the instruction type. For instructions with all register operands, typically the destination register is listed first, followed by the source operands. This is the typical format for arithmetic and logical instructions. Note that `nop` is a pseudo-instruction, requiring no operands, which is implemented as an `addi $x0, $x0, 0` instruction. In the example code Listing 5, line 3 shows an example `add` instruction with source registers `$x0` and `$x3` and destination register `$x5`.

Memory instructions (loads and stores) use a format in which the address register is surrounded with square brackets, and the offset or offset register is listed before the brackets. This syntax is similar to the form seen in the RISC-V ISA document and in assembly dumps, thought the ISA document uses parenthesis rather than square brackets. Listing 5 line 4 shows an example of this syntax. In that example, the address in register `$x5` is added to the offset of 0, and the value at that location is written to destination register `$x9`.

Table 2. Instruction mnemonics recognized by the *dt* high-level assembler

Arithmetic	Memory	CTI	Other
lui	lb	jal	nop
auipc	lh	jalr	fence
addi	lw	beq	fence.i
slti	lbu	bne	ecall
sltiu	lhu	blt	ebreak
xori	sb	bge	csrrw
ori	sh	bltu	csrrs
andi	sw	bgeu	csrrwi
slli		j	csrrsi
srli		jr	csrrci
srai		ret	
add			
sub			
sll			
slt			
sltu			
xor			
srl			
sra			
or			
and			
mul			
div			

```
1 mem (0x000000400000) {
2         # ...
3         add $x5, $x0, $x3
4         lw $x9, 0[$x5]
5         beq $x9, $x0, target
6         # ...
7 target: nop
8 }
```

Listing 5. Code example of several instructions

Control transfer instructions have no destination register, so their source register(s), if any, are listed immediately after the mnemonic. For the instructions that allow for direct targets (either immediate addresses, or offsets from the PC), the *dt* high-level assembler allows you to use a labeled memory location in place of the offset. In that case, *dt* will determine the appropriate address or offset automatically. Listing 5 line 5 shows that a `beq` instruction will skip over some code to a `nop` named `target` if the `beq` is taken. Table 2 also lists a `ret` instruction, which is a pseudo-instruction for a `jalr $x0, 0($x1)`. The `ret`, if used, requires no operands.

3.3. High-Level Language Syntax

A programmer can simply use the syntax from the previous sections to write RISC-V programs entirely in assembly language syntax. However the real strength of *dt* is in the additional high-level language syntax that can be intermixed with assembly syntax. This section describes these additional high-level language features, and outlines the exact instructions that will be emitted by `dt` when using high-level

Operation Format	Resulting Instruction
reg = *reg* + *reg*	`add`
reg = *reg* + *imm*	`addi`
reg = *reg* - *reg*	`sub`
reg = *reg* - *imm*	`addi`
reg = *reg* * *reg*	`mul`
reg = *reg* / *imm*	`div`
reg = *reg*	`addi`
reg = *imm*	See discussion
reg = -*reg*	`sub`
reg = *reg* & *reg*	`and`
reg = *reg* & *imm*	`andi`
reg = *reg* $\vert$ *reg*	`or`
reg = *reg* $\vert$ *imm*	`ori`
reg = *reg* ˆ*reg*	`xor`
reg = *reg* ˆ*imm*	`xori`
reg = *reg*	`xori`
reg = *imm*	Not yet implemented
reg = *reg* << *imm*	`slli`
reg = *reg* << *reg*	`sll`
reg = *reg* >> *imm*	`srli`
reg = *reg* >> *reg*	`srl`
reg = *reg* < *reg*	`slt`
reg = *reg* < *imm*	`slti`
reg = *reg* > *reg*	Not yet implemented
reg = *reg* > *imm*	Not yet implemented
reg = *reg* <= *reg*	Not yet implemented
reg = *reg* <= *imm*	Not yet implemented
reg = *reg* >= *reg*	Not yet implemented
reg = *reg* >= *imm*	Not yet implemented
reg = *reg* == *reg*	Not yet implemented
reg = *reg* == *imm*	Not yet implemented
reg = *reg* != *reg*	Not yet implemented
reg = *reg* != *imm*	Not yet implemented
reg = @*label*	See discussion (address-of operator)

Table 3. Assignment operations allowed by *dt*

syntax.

3.3.1. Assignments. Many common operations have an alternative shorthand notation which is similar to the C set of operations. The general form is to list a destination register (or named register), followed by one of the operands and operators listed in Table 3. The table also shows which instruction will be used to implement the assignment – the destination register will always use the register listed on the left-hand side of the assignment, and the source operands will always use the register/immediate on the right-hand side. If two registers appear on the right-hand side, then they will be used in the same order as they appear in the expression.

Note that only a single operation can be done per assignment, compound operations, or operations on three or more operands are not allowed. This is because the *dt* high-level assembler does not do register allocation. So, the programmer must use separate lines for each intermediate result, explicitly identifying which registers should be used. This also means that the order of operations is irrelevant.

Some operations listed in Table 3 require more than one instruction, or require one of many possible different instructions. Assigning an integer register to another integer register is simply a case of using an `addi` with zero immediate to do the copy. But when assigning an immediate value to a register, depending on the size of the immediate, the operation may be done with a single `ori`, or may require an `lui` followed by an `ori`.

Some assignments require a little creativity. For example, to assign the negated value of a register to another register, *dt* will use a `sub` where the first operand is the sink register `$x0` and the second is the register on the right-hand side of the assignment.

Several of the comparison operators have not yet been implemented. This is because RISC-V does not have single instructions to perform the comparison. These operations will require several instructions each, and is left for future work.

`dt` permits one type of assignment that is treated specially. The `$pc` register refers to the machine program counter (instruction pointer) register. A programmer can assign an immediate value to `$pc`. This does *not* produce any instruction, instead it is used to specify the program entry point, to be added to the metadata written to the output of `dt` (depending on the type of output). It is required to put the assignment of `$pc` outside of any `mem()` blocks. If no assignment to `$pc` is used, then the entry point is assumed to be address `0x000000000000`.

Listing 6 shows two `mem()` blocks with equivalent instructions. However, one `mem()` block is written using the instruction mnemonics, and the other is written using the shorthand assignments. These two `mem()` blocks will produce binary equivalent instructions. Note that these `mem()` blocks could not appear in the same program, as their memory regions would overlap. Also note that code block (b) shows the `$pc` assignment that shows explicitly shows that the program entry point is address `0x10000`.

```
# code block (a)
mem (0x10000) {
    ori  $x1, $x0, 3
    xori $x1, $x1, -1
    and  $x3, $x2, $x1
    slt  $x4, $x3, $x0
}

# code block (b)
$pc = 0x10000

mem (0x10000) {
zero: $x0
mask: $x1
val:  $x2
res:  $x3
cond: $x4

      mask = 3
      mask = ~mask
      res = val & mask
      cond = res < zero
}
```

Listing 6. Code example of equivalent instructions using (a) instruction mnemonics and (b) shorthand assignments

Another useful feature of *dt* is the support provided for an address-of operator, using the `@` symbol. This can be used to assign the address of any named memory address, whether it is a labelled instruction definition or a data value. Listing 7 shows two different uses of the address-of operator – the first to easily read a data value from memory, and the second to get the address of an instruction to be used as the target of a jump.

```
mem (0x10000) {
    $t0 = @value      # get the address of literal "value" 123
    lw $t1, 0[$t0]    # read memory to get the value
    $t2 = @loop       # get the address of jr pseudo-instruction

loop:
    jr $t2            # infinite loop to end program

value:
    .word 123
}
```

Listing 7. Code example showing uses of the address-of operator

3.3.2. Block Statements. The last group of syntatical constructs provide the high-level language-like features of if-statements and loops. The syntax for each of these constructs is similar to the C programming language. However, the major difference is that the condition must be a single register or named register. This is because a complex condition requires a temporary register, and the *dt* high-level assembler does not do register allocation. This also eliminates the `for` loop from availability: the initial value, and increment amount could be handled, but the comparison to know when the loop should stop requires a register. The *dt* high-level assembler also has no formal mechanism or syntatical construct for functions. This is due to the several requirements that functions require, a runtime stack, the stack pointer, the return register, function arguments, and so on – all of these are against the intent behind *dt* to give all control to the programmer.

The constructs that are available however, are: `if`-statements, `if-else` statements, `while` loops, `do`...`while` loops, `until` loops (similar to `until` loops in BASIC or scripting languages), and `do`...`until` loops. These constructs can contain instructions, definitions, values (if you really want to mix instructions with data), assignments, and other block statements. These block statements can also be named themselves – simply provide a label followed by a colon followed by the block statement. This will name the first instruction of the block statement. That named instruction might be an instruction in the body of the block statement as in `do`...`while` and `do`...`until` loops, or it might be an instruction that is not evident in the code (i.e. part of the supporting code emitted by the block statement).

The condition for each of the statements must be an integer register or label that corresponds to an integer register. The meaning, however, is the same as in C – any non-zero value is considered `true`, and zero is considered `false`. Table 4 gives the syntax for each of the constructs and the code generated by the *dt* high-level assembler.

3.4. Code Examples

This section includes two example programs written in *dt*. The intent is to show longer, full program examples, and also to show the flexibility of the syntax.

dt Syntax	Assembly Produced

if (reg) {
    # code body
}
	beq reg, $x0, label1
	    # code body
	label1:

if (reg) {
    # code body
}
else {
    # code body
}
	beq reg, $x0, label2
	    # code body
	    j label3
	label2:
	    # code body
	label3:

while (reg) {
    # code body
}
	beq reg, $x0, label4
	label5:
	    # code body
	    bne reg, $x0, label5
	label4:

until (reg) {
    # code body
}
	bne reg, $x0, label6
	label7:
	    # code body
	    beq reg, $x0, label7
	label6:

do {
    # code body
} while (reg)
	label8:
	    # code body
	    bne reg, $x0, label8

do {
    # code body
} until (reg)
	label9:
	    # code body
	    beq reg, $x0, label9

Table 4. Structured control flow code blocks recognized by *dt*

3.4.1. Bubble Sort. This first example shows an implementation of the bubble sort algorithm. The first `while` loop populates memory, starting at address `0x000000000000`, with arbitrary values in descending order. The second (nested) `while` loop implements the actual bubble sort, and sorts values into ascending order. This second loop saves the sorted array in place. The final `while` loop iterates through the array to verify that the order is correct, counting the number of correct positions, and saving that count to memory address `0x000000600000`.

```
$pc = 0x000000400000

mem (0x000000400000) {
ii:      $x1
max:     $x3
addr:    $x2
data:    $x4
cond:    $x5

flag:    $x6
val1:    $x7
val2:    $x8
comp:    $x9
result:  $x10
jj:      $x11

    max = 512  # number of elements

    # fill with values
    addr = 0x0
```

```
    data = 0x041ab25e
    ii = 0
    cond = ii < max
    while (cond) {
        sw data, 0[addr]
        data = data - 3
        addr = addr + 4
        ii = ii + 1
        cond = ii < max
    }

    flag = 1

    # perform sort
    while (flag) {
        flag = 0
        addr = 0x0
        ii = 0
        cond = ii < max
        while (cond) {
            lw val1, 0[addr]
            lw val2, 4[addr]

            comp = val2 < val1
            if (comp) {
                flag = 1
                sw val2, 0[addr]
                sw val1, 4[addr]
            }

            addr = addr + 4
            ii = ii + 1
            cond = ii < max
        }
    }

    # verify sorted result
    addr = 0x00000000
    result = 0
    ii = 0
    cond = ii < max

    while (cond) {
        lw val1, 0[addr]
        lw val2, 4[addr]

        comp = val1 < val2
        if (comp){
            result = result + 1
        }
        addr = addr + 4
        ii = ii + 1
        cond = ii < max
    }

    addr = 0x000000600000
    sw result, 0[addr]

inf:    j inf
}
```

Listing 8. Bubble sort example source code

3.4.2. Matrix Multiply. The second example program is used to compute a matrix multiplication of two matrices, call them **A** and **B**, and write the result to matrix **C**. The matrices are 4 rows by 4 columns and their data is initialized in a separate `mem()` block from the instructions and are saved in row-major form as would have been done by a C compiler. The code is shown in Listing 9.

```
$pc = 0x000000400000

mem (0x000000400000) {
ii:      $x1
jj:      $x2
kk:      $x3
icond:   $x4
jcond:   $x5
kcond:   $x6
aaddr:   $x7
baddr:   $x8
caddr:   $x9
aval:    $x10
bval:    $x11
cval:    $x12
mtemp:   $x13
stemp:   $x14
four:    $x15
sixteen: $x16
mres:    $x17

    four = 4
    sixteen = 16
    ii = 0
    icond = ii < 4
    while (icond) {
        jj = 0
        jcond = jj < 4
        while (jcond) {
            # initialize c[i][j] to zero
            cval = 0

            kk = 0
            kcond = kk < 4
            while (kcond) {
                # compute address of a[i][k]
                aaddr = @A
                mres = ii * sixteen
                mtemp = mres
                mres = kk * four
                stemp = mres
                mtemp = mtemp + stemp
                aaddr = aaddr + mtemp

                # compute address of a[k][j]
                baddr = @B
                mres = kk * sixteen
                mtemp = mres
                mres = jj * four
                stemp = mres
```

```
                mtemp = mtemp + stemp
                baddr = baddr + mtemp

                # load the values of a[i][k] and b[k][j]
                lw aval, 0[aaddr]
                lw bval, 0[baddr]

                # c[i][j] += a[i][k] * b[k][j]
                mres = aval * bval
                mtemp = mres
                cval = cval + mtemp

                kk = kk + 1
                kcond = kk < 4
            }

            # compute c[i][j] address
            caddr = @C
            mres = ii * sixteen
            mtemp = mres
            mres = jj * four
            stemp = mres
            mtemp = mtemp + stemp
            caddr = caddr + mtemp

            # store c[i][j]
            sw cval, 0[caddr]

            jj = jj + 1
            jcond = jj < 4
        }
        ii = ii + 1
        icond = ii < 4
    }
}

# matrix data
mem (0x000010000000) {
A :  .word 10   # A[0][0]  0x000010000000
     .word 2    # A[0][1]  0x000010000004
     .word 7    # A[0][2]  0x000010000008
     .word -4   # A[0][3]  0x00001000000c
     .word 9    # A[1][0]  0x000010000010
     .word -2   # A[1][1]  0x000010000014
     .word 12   # A[1][2]  0x000010000018
     .word 1    # A[1][3]  0x00001000001c
     .word 17   # A[2][0]  0x000010000020
     .word 8    # A[2][1]  0x000010000024
     .word -3   # A[2][2]  0x000010000028
     .word 1    # A[2][3]  0x00001000002c
     .word 6    # A[3][0]  0x000010000030
     .word -5   # A[3][1]  0x000010000034
     .word 13   # A[3][2]  0x000010000038
     .word 0    # A[3][3]  0x00001000003c

B :  .word 3    # B[0][0]  0x000010000040
     .word 7    # B[0][1]  0x000010000044
     .word 9    # B[0][2]  0x000010000048
     .word -2   # B[0][3]  0x00001000004c
     .word 15   # B[1][0]  0x000010000050
```

```
        .word  -1    # B[1][1]    0x000010000054
        .word  4     # B[1][2]    0x000010000058
        .word  6     # B[1][3]    0x00001000005c
        .word  11    # B[2][0]    0x000010000060
        .word  8     # B[2][1]    0x000010000064
        .word  3     # B[2][2]    0x000010000068
        .word  -7    # B[2][3]    0x00001000006c
        .word  1     # B[3][0]    0x000010000070
        .word  10    # B[3][1]    0x000010000074
        .word  4     # B[3][2]    0x000010000078
        .word  -5    # B[3][3]    0x00001000007c

C :     .word  0     # C[0][0]    0x000010000080
}
```

Listing 9. Matrix multiplication example source code

4. Output File Formats

Out-of-the-box dt provides its output in a variety of formats. This section outlines each of those formats. The intent of supporting several output formats is to maximize flexibility for the users of dt to use the dt output as the input to other tools – simulators, emulators, kernels, virtual machines, compilers/assemblers, Verilog testbenches, FPGA block memories, and so on.

4.1. Checking Output Format

When issuing the dt command, using the -checking command-line option will emit machine code and a variety of debugging information to stdout. This output could be redirected to a file for future use, or can simply be used for debugging purposes.

When using the -checking output, three key pieces of information are displayed: the entry point (memory address) of the program, the instructions/data for each mem() block, and the symbol table. All of these items are emitted, even if the .dt program does not have syntax to modify one of them (for example, the PC will be displayed, even if the program never changes the default value of $pc).

The program entry point is displayed using the text "Program counter:", a tab character, followed by the entry point address. The memory address of the entry point will be printed in hexadecimal, preceded by the C-style 0x, and will have 48-bits (12 hex digits), zero padded if needed. Displaying only 48-bits for addresses is used throughout dt output, as the use of the full 64-bit address space is uncommon.

Field	Output	Origin
<type>	inst:	An instruction
	bdata:	A .byte directive
	hdata:	A .half directive
	wdata:	A .word directive
	ldata:	A .long directive
	fdata:	A .float directive
	ddata:	A .double directive
	sdata:	A .stringz directive
	def:	A register label definition
	join:	A join-node
<addr>	A 48-bit address, in hexadecimal	The memory address of this line, if one exists
<value>	An *n*-bit value, in hexadecimal	For instructions *n*=32 bit, the encoding
		For data values *n*=8, 16, 32 or 64 depending on the directive
	A string	For the .stringz directive, strings will be surrounded by double-quotes
	A label	For join-nodes and register definitions
<asm>	The instruction, in assembly language source	Only exists for lines that correspond to instructions

Table 5. Meaning of each field of a line of mem() block information when using -checking

For each mem() block in the .dt program, the -checking output will print a line mem() block: <address>:, where <address> will be the starting address of the mem() block. This header line will be followed by several lines, each of which corresponds to the contents of the mem() block. The lines after this header have the format: <type> <addr> <value> <asm>. Each of these fields are separated by a tab character. Table 5 describes each of these fields. The **Field** column indicates which field the row describes, the **Output** column describes what can possibly appear for that field, and Origin describes what .dt code produces that type of field.

After each mem() block has been output, the last piece of information shows the symbol table that was built when assembling the .dt program. There will always be a header row with the exact text: Symbol table entries:. Thereafter, there will be one row for each symbol that appears in the .dt program. Each row will have entry[<n>]:, where n increments for each additional symbol in the .dt program. After the entry number, is the symbol itself – whether a symbol indicated by the programmer, or an internally generated symbol (for join-nodes). The next field will indicate the type of symbol, either mem for a labelled memory location, or reg for a labelled register. The last field of each entry will be either an address, in 48-bit hexadecimal, for named memory locations, or the register number for named registers. The register names will always use the format $x<n> where <n> refers to the register number.

4.2. ELF64 File Format

dt is able to produce a Linux executable when using the -elf command-line flag. It is well beyond the scope of this technical report to describe the ELF64 file format. An interested reader can either refer to the elf man page, or the ELF64 standard [1] for more in-depth information on the ELF64 file format. This section will only describe the properties of the executable emitted by dt. The executable produced by dt has been tested to execute using the HiFive Unleashed [2] by SiFive, running Debian Linux.

The executable produced by dt is the bare minimum to support execution in Linux. The executable is always statically linked and consists of an ELF header, a program header table, and a section.

The ELF header fields are mostly hard-coded to match the needs of a RISC-V executable. The e_machine is set to EM_RISCV, e_type set to ET_EXEC, e_ident[EI_CLASS] set to ELFCLASS64, e_ident[EI_DATA] set to ELFDATA2LSB, and so on. The only surprise is the .dt must set the $pc entry point to 0x000000400000, which is then internally adjusted, since the entire program executable will be loaded to memory, the actual first instruction appears at a later address (based on the size of the ELF header plus the size of the program header table).

The program header table itself is an array of meta-data, one element for each loadable section. In dt, it is currently assumed there is only one loadable section. Thus, there is only a single entry in the program header table, which refers to a single mem() block of a .dt program. This program header table entry p_type is marked as PT_LOAD, as this single mem() block should be loaded into memory by the Linux loader. The section will have all permissions (read, write and execute), and so the program header table entry p_flags will be set to allow all permissions. This means that the single mem() block, though likely executable instructions, could have its memory locations overwritten (i.e. to permit self-modifying code). It is permissible to mix instructions and data in this single mem() block of the .dt program. However it is up to the .dt programmer to ensure that data isn't accidentally executed.

The section data copied into the remainder of the ELF executable output file is exactly the instruction encodings and/or data from the single mem() block of the .dt program. Note that dt will pad data values with zero values to enforce correct alignment based on the data type (4 bytes for instructions, 2 bytes for .half, 4 bytes for .word, etc.).

Traditionally, a full Linux executable (even statically linked) will also include sections for things like debugging information, a section header table, string tables, and so on. However, these sections are optional per the ELF64 standard, and have been omitted to the bare minimum needed for correct execution in Linux.

4.3. Flat Memory Image File Format – Text

A full Linux executable may be overkill for many users of `dt`, so the `-text` (this section) and `-bin` (next section) command-line options were introduced to permit output in simplified output that can easily be read as input to other tools.

The `-text` output produces an ASCII-encoded text file, rather than printing to `stdout`. The file format has minimal meta-data and is intended to display data as a flat memory address image. There are no restrictions on the number of `mem()` blocks, or the entry point address of the `.dt` program. The output was formatted to loosely resemble the UNIX `xxd` tool, a command-line hex editor.

Values (instruction encodings or data) from each `mem()` block will appear with 16 bytes worth of values per row of the output, in hexadecimal. The values will 16 byte aligned, padded with zeros if necessary, and each byte separated by a space character. For values that consist of multiple bytes (half words, words, machine instructions, etc.) the bytes are saved in little-endian order.

Each line of output will have a 48 bit address, in 12 hex digits, as the first column of the output. The output is not guaranteed to be in address order, the addresses will appear in the same order as the `mem()` blocks appear in the `.dt` program. The columns of addresses/values are separated by spaces. Each `mem()` block will be followed by a blank new line.

The `-text` output does not show the program entry point, nor any of the labels from the symbol table.

Listing 10 shows a simple `.dt` program with two `mem()` blocks, one with a single instruction, and one with a single data value. The output that will result is shown in Listing 11. Notice that the `mem()` block with the data value is not aligned to a 16 byte boundary, but the output has been automatically adjusted accordingly. Also, both values (the instruction, and the word) are 4 byte quantities, and are in little-endian order.

```
mem(0x100000000008){
    .word 0xdeadbeef
}

mem(0x000000400000){
    addi $x1, $x0, 99
}
```

Listing 10. Code example used to produce a text file output

```
100000000000   00 00 00 00 00 00 00 00 ef be ad de 00 00 00 00

000000400000   93 00 30 06 00 00 00 00 00 00 00 00 00 00 00 00
```

Listing 11. Result of running `dt` on Listing 10 when using `-text` output

4.4. Flat Memory Image File Format – Binary

The goals for the flat binary memory image file format is similar to those of the text file format – the file format should be simple, and express as much of a *dt* program as possible. To have `dt` emit the flat binary memory image, use the `-bin` command-line option.

Each `mem()` block of a *dt* program will be numbered, such that the top-most `mem()` block in the source code will be number 0, the next will be number 1, and so on. When emitting the binary flat memory image, `dt` will write each `mem()` block to their own file. This was done to minimize file meta-data. Files will be given the base name (just `a` by default, or the base name provided at the command-line with the `-out` option), and that base name will immediately be followed with a dash and the `mem()` block number, then ending with the `.bin` file name extension. For example, a *dt* program consisting of two `mem()` blocks, using the default file names would be `a-0.bin` and `a-1.bin`

Within the files, the only meta-data will occur at offset zero in the file, and will be eight bytes – the starting address of the `mem()` block contained within the file. These addresses are automatically adjusted to 16 byte alignment.

After the starting address, the remainder of the contents of binary memory image files will be the data of the `mem()` block. The data will be padded out to 16 byte alignment. Bytes used for padding will have the value zero. Finally, the multi-byte data is saved in little-endian order.

5. `dt` Source Code

This section briefly describes the source code for the `dt` high-level assembler itself. The source is written using *flex* and *bison* for the lexer and parser, and the rest of the code is written in C. One strength of `dt` is that the code is relatively short, with only roughly 3200 SLOC (generated using David A. Wheeler's 'SLOCCount'). Thus, as long as a programmer is already familiar with *flex*/*bison*/C, the code base can be understood in a short period of time.

`dt` is a two-pass assembler. The first pass scans and parses the `.dt` program, and builds up several data structures. There are cases when not all information is known, however. For example, during the first pass, a labelled target may not have been encountered when a *use* of that label is found. After the first pass, all labels will have been scanned, and so the second pass iterates through the data structures from the first pass to fill in these details and then emit the final output.

The discussion of the code will be split into two sub-sections, the first will give a high-level view of the directory tree, and the second will describe some of the data structures and functions that were written to implement `dt`.

5.1. `dt` Files

All source code, including the header files, for `dt` appear in the `src/` subdirectory under the top-level directory. Once compiled, their object files will be saved to the `obj/` subdirectory, and the overall `dt` executable will appear in the `bin/` subdirectory. Table 6 outlines the contents of each of the source code files.

File	Purpose
`dt.l`	Scanner/lexical analyzer.
`dt.y`	Contains the `main()` entry point, as well as the parser.
`riscvarch.h`	Holds constants that describe RISC-V ISA-specific values like opcodes and function codes.
`inst.c/.h`	Holds the main `instruction_t` data type which defines a RISC-V instruction. Also implements the functions that manipulate instances of `instruction_t`.
`mem.c/.h`	Defines a handful of data types and functions used to maintain lists of instructions/data. Although the nodes in these lists represent entities that occupy memory (of the target RISC-V machine), there are some cases where these files represent nodes that do not require any memory.
`pc.c/.h`	Holds the `.dt` program entry point address.
`symtab.c/.h`	Implements the symbol table.
`output.c/.h`	Functions used to emit the final `.dt` program in any of the supported output formats.
`util.c/.h`	Implements the `dt` version number, as well as some of the generic support functions.

Table 6. Source code files used to implement `dt`

5.2. `dt` Data Structures and Functions

Throughout `dt` are a handful of key data structures and functions that carry out the bulk of implementation. This section covers some (not all) of these data structures and functions, to make reading the actual `dt` source code a little quicker for a developer.

5.2.1. Instructions. The main goal of `dt` is to emit instructions, and it thus follows that the instruction representation is key. The `instruction_t` implements an instruction, and its declaration can be found in `inst.h`. This data type has member variables for representing the instruction type in two ways: a shorthand implemented by the `inst_id`, a unique number for each instruction mnemonic, and also by the longer combination of the `opcode`, `funct3` and `funct7` member variables. The latter of these are the actual encodings that will be used when emitting the binary instruction encoding.

There are several `unions` that are also used to hold binary values that will appear in the final instruction encoding. The `union` is used to encompass any bit field that overlaps in the RISC-V instruction encoding type (R-type vs. I-type vs. S-type, etc.). The last member variable, called `target_name` is a string that is used to hold the target label when the `.dt` program uses a target label instead of a target address.

The `calculate_offsets()` function also supports these instructions that use a target label. Once all addresses are known (calculated in the parser), `calculate_offsets()` finds the label in the symbol table, and uses the corresponding memory address to calculate memory offsets.

The `encode_*_type()` family of functions actually carries out the bitwise encoding of each of the instruction types.

5.2.2. `mem()` Block Representation. Each `mem()` block in a `.dt` program is represented with a node of a linked list. The nodes in this list are of type `struct memblock_list_type`/`memblock_list_t`. Each node has member variables for the highest and lowest address used within the `mem()` block it represents. Each node also has a pointer to the instructions/definitions/values found within that `mem()` block.

One important function that is related to `mem()` blocks is in the `check_mem_bounds()` function. This function iterates through all `mem()` blocks (i.e. each `memblock_list_t` node of the list), comparing the lowest and upper-most address of all blocks to make sure none of the blocks have overlappin addresses.

Since the contents of a `mem()` block could be one of a handful of categories (an instruction, a data value, or a definition), there is a layer of indirection in the `struct mem_entry_type`/`mem_entry_t`. Instances of `mem_entry_t` are also linked list elements. The `type` member variable describes whether the node is for an instruction, a data value, or a definition (as defined in the `type_t`) enum. The `type` describes the difference between an instruction, a data value, or a definition. However `type_t` also indicates that the node type could be a join-node. A join-node is a dummy node (i.e. does not represent any line of code from the original `.dt` program) that appears after the closing curly brace of a loop or `if`-statement. Thus, a join-node is the point at which control flow reconverges in a control flow graph. Since joinnodes represent targets of instructions that are automatically inserted by `dt` for high-level language statements, they have not been given a label by the `.dt` programmer. To solve this issue, target names of join-nodes are randomly generated internally. The function `internal_name()` generates these names.

The `name` member variable of a `mem_entry_t`, a string, is used whenever the node is representing a line of `.dt` code that has been labelled. The `status` member variable can take one of two (enum) values, `ENTRY_COMPLETE` or `ENTRY_INCOMPLETE`. A node that is incomplete is one that uses a target label, but the label does not yet have an address (i.e. the first pass of the assembly process).

Listing 12 presents a hypothetical `.dt` program that has several `mem()` blocks, each of which contain different types of code. Figure 2 shows the data structures used to represent this hypothetical program. The first `mem()` block contains only definitions for registers, which is shown on the right-most `memblock_list_t` entry in Figure 2.

```
1  mem(0x00000000){
2  val:   $t0
3  addr:  $sp
4  }
5
6  mem(0x00400000){
7  first: lw  val, 0[addr]
8          if (val) {
9              sw val, -4[addr]
10         }
11 }
12
13 mem(0x10000000){
14     .word 0xdeadbeef
15 }
```

Listing 12. Code to highlight what data structures will be produced

The second `mem()` block in Listing 12 has a short instruction sequence, including an `if`-statement. Notice the right-most `memblock_list_t` entry in Figure 2. It refers to four `mem_entry_t` instances, the top-most holds the properties of the `lw` instruction from line 7. The next `mem_entry_t` represents the `beq` that implements the `if`-statement, which will target the join-node at the bottom of the list of `mem_entry_t` instances. Since there is no instruction after the `if`-statement, the join-node still correctly allows *dt* to produce the correct offset for the `beq` instruction. If there *was* an instruction after the `if`-statement body, there would still be a join-node in addition to the instruction node after the join-node... they would simply have the same address.

The last `mem()` block in Listing 12 holds only a single data value, and is represented by the middle node of the list of `memblock_list_t` instances.

5.2.3. Symbol Table. `dt` must implement a symbol table, since labels are permitted. Traditionally, the symbol table is a key-value pair where the key is the label, and the value is the memory address named by that label. However, since *dt* permits registers to also be named, the symbol table in `dt` must accommodate this additional need.

The symbol table in `dt` is also implemented as a linked list, with nodes of type `struct symtab_entry_type/symtab_entry_t`. The difference between a named memory location versus a named register is differentiated by the `type` member variable. The `name` string holds the name of the definition, and `value` holds either the address (for named memory locations) or register

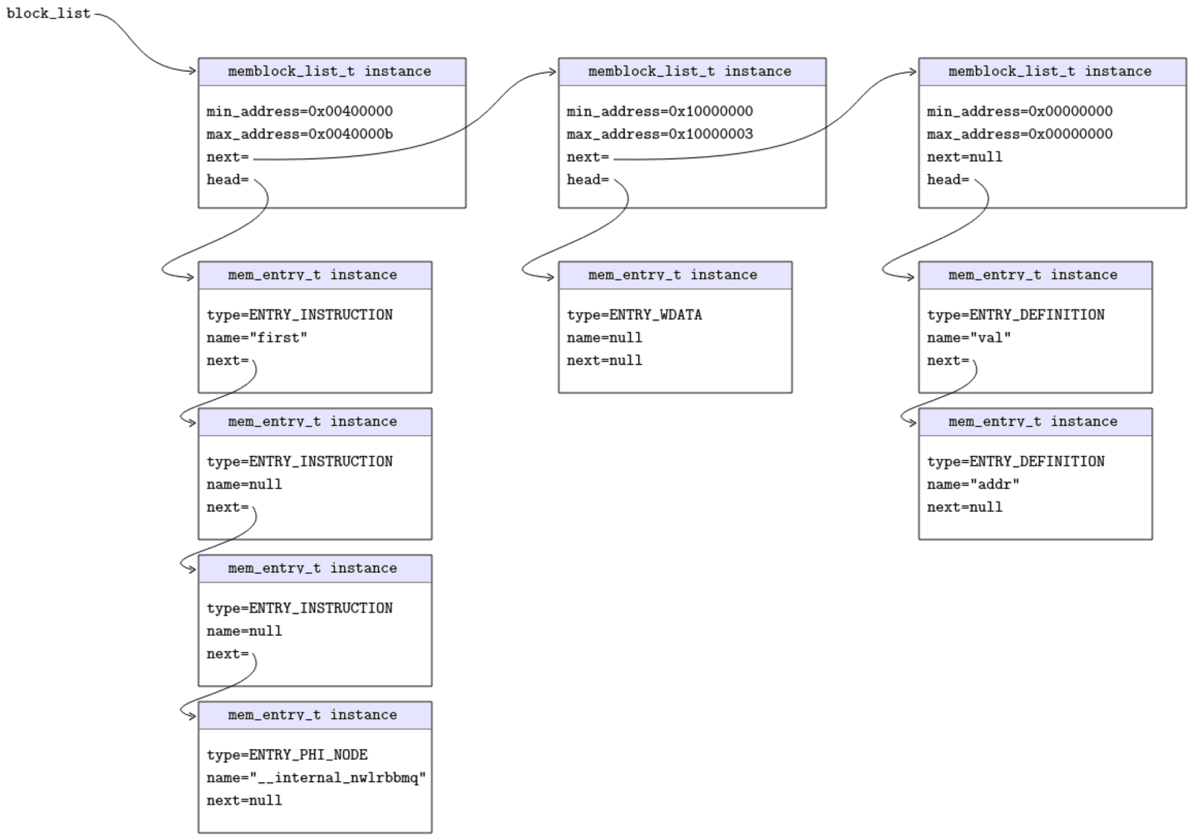

Figure 2. Internal data structure representation of an example `.dt` program.

number (for named registers).

Continuing the example program from Listing 12, the resulting symbol table would have four nodes, two for each of the named registers (`"val"` corresponds to register `$t0`, and `"addr"` for register `$sp`). The first instruction of the program (the `lw`) is labeled `"first"` and has the address `0x00400000`, and finally, the join-node after the `if`-statement has the name `"__internal_nwlrbbmq"` and technically does have an address of `0x0040000c`.

6. Known Bugs and Limitations

Both the *dt* language and the `dt` high-level assembler are works in progress. In [3], we outline ideas for features that we would like to incorporate into future versions of DuctTape. However, we also acknowledge that there are existing features that are less than perfect or have outright problems at the time of this publication. The following lists these known limitations.

1) The starting address of ELF64 executables must be `0x000000400000`. But this is not checked by `dt` when assembling.
2) There is no way to have more than one `mem()` block in ELF64 executables. Thus, it is not possible to initialize memory locations outside of the expected *.text* segment. To maximize the utility of *dt*, it should be possible to arbitrarily add additional loadable segments to the ELF64 output.
3) There has been some forethought to eventually support RV64F and RV64D floating-point instructions. For example, there are assembler directives for initializing memory locations with IEEE 754 32- and 64-bit floating-point values. However, the scanner does not recognize the floating-point instructions or the floating-point registers. The existing `.float` and `.double` directives have not been tested at all.
4) `dt` does do some sanity-checking, for example checking that `mem()` blocks do not have addresses that overlap. However there is no sanity-checking of bit widths. Several machine instruction encodings have bit fields for immediate values and offsets, and currently these can be specified when writing the instructions in assembly. But `dt` currently truncates bit values if the `.dt` programmer accidentally uses a value requiring more bits than permitted by the instruction encodings.
5) *dt* currently has no way to define constants. A *dt* programmer can set a memory location aside, then read that memory location with a load instruction. But there is no way to define a label for an immediate value or offset.

References

[1] "Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification," 1995, Specification. [Online]. Available: https://refspecs.linuxbase.org/elf/elf.pdf

[2] "HiFive Unleashed," 2020, Product Brief. [Online]. Available: https://www.sifive.com/boards/hifive-unleashed

[3] J. Severeid and E. Forbes, "dt: A High-level Assembler for RISC-V," in *Proceedings of the 53rd Midwest Instruction and Computing Symposium*, April 2020.

[4] A. Waterman and K. Asanovic, "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191214-draft," December 2019, Manual. [Online]. Available: https://riscv.org/specifications/