dt: A High-level Assembler for RISC-V

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

Abstract

The RISC-V instruction set is a modern RISC instruction set specification that has gained attention from academics, hobbyists and even major industry players due to its open-source license. Anyone can build simulation tools, and even full hardware designs, targeting the RISC-V instruction set. During the early development stages of these tools and designs, it is important to be able to exercise specific instruction sequences (micro-kernels) for testing corner-cases, to expose difficult-to-reproduce bugs, or to characterize and verify performance estimates.

This paper describes a new programming language intended to make it easier to write micro-kernels for the RISC-V instruction set. The programming model is strictly at the assembly level, however there are high-level language constructs that are available to ease the programming burden. The language is called DuctTape, or simply dt *for short. This style of programming enables the programmer to have complete control over the emitted instructions and memory values without the burden of writing code entirely in assembly. The output of the* dt *high-level assembler is either 1) a flat memory space file that can easily be integrated into processor simulators or FPGA designs, or 2) the ELF64 format, suitable for executing on a RISC-V Linux development platform.*

1. Introduction

DuctTape (stylized as *dt*) is a programming language and assembler for the RISC-V instruction set. It differs from standard assembly language and the GNU tools, however, by providing several syntactical constructs borrowed from high-level programming languages such as symbolic operators, assignments, loops and `if`-statements. The result is a hybrid language that is part assembly and part high-level language.

The motivation for such a language and assembler was realized when trying to debug and verify Verilog hardware implementations of full processors [14] both for correctness and for performance. The need to carefully craft instruction sequences to either avoid known bugs or to explicitly exercise and test corner-cases was evident, leading developers to write micro-kernels – simple, but specific sequences of instructions. These same needs exist when developing a variety of machine-specific technologies, including emulators, simulators, kernels, run-time environments and virtual machines.

The GNU compiler [10] (*gcc*) and C programming language can be used to write short test programs, but the programmer does not have complete control over several aspects: the registers that the compiler uses, the memory locations used, the types and order of instructions emitted, the format of the output file, and so on. If library code is used, then these problems may be difficult to rectify – possibly requiring library code to be recompiled. Additionally, it can be difficult to completely avoid all library code, since code that executes before/after the `main()` function is automatically inserted.

The GNU assembler (*gas*) and assembly programming language can be used to regain control over some aspects of the instructions produced. But some of the limitations listed for *gcc* above still persist. In fact, assembly language and *gas* only provides the additional control over the registers used, and the types and order of instructions emitted. Using *gas* also puts a greater burden on the programmer, for example requiring setting up the equivalent of structured control statements by hand, and requiring the programmer mentally keep track of which registers are used for which purposes.

Contrasting *dt* with *gcc* and *gas*, *dt* allows the programmer very tight control over the emitted code. The programmer can write programs strictly using assembly language, but can also use several high-level language constructs or even intermix assembly with high-level constructs. For example, *dt* language allows the programmer to easily give names to both registers and memory locations, assign specific addresses to data and instructions, use assignment statements with arithmetic/logic operators, use `if`-statements and `while`-loops, and can even intermingle data among the program instructions. To keep the instruction-level control over the emitted code, the high-level constructs provided were selected such that they all result in deterministic instruction sequences with architecturally-visible side-effects.

These code constructs can drastically increase the productivity of the micro-kernel programmer, while still giving them complete control of instructions and data. For developers of simulators, emulators and run-time environments, hardware, virtual machines, and operating system kernels the *dt* language and assembler can help in tracking down rare or specific bugs, and can help validate correctness and performance characteristics.

Intended to be as flexible as possible in order to support all of these development efforts, the output file format of *dt* comes in two main forms. First is a binary or text file with a flat memory image that can easily be read directly into simulators or hardware designs – for example into simulation testbenches, or into hardware block memories of Field Programmable Gate Arrays (FPGAs). The second possible output

of *dt* is a statically-linked ELF64 Linux executable file. For validating the output of *dt* itself, we used the ELF64 executable output to run natively on the HiFive Unleashed development board [2].

The remainder of this paper covers many aspects of the *dt* toolchain and language. Section 2 provides some background on RISC-V, the various tools targeting RISC-V, and prior art on high-level assemblers. Section 3 outlines the basic structure of *dt* programs, the high-level language features it provides (including a rationale for excluding other features), and how those high-level features produce deterministic instruction sequences. Examples of subtle hardware bugs, and how *dt* can help avoid those bugs are outlined in Section 4. Section 4 also shows how *dt* could be used to validate performance. We conclude in Section 5, where we also outline the current status of *dt*, ideas for future work, and point out possible use-cases for *dt* in an academic setting. A separate technical report [16] serves as a full language reference, including the exact code that will be emitted for each of the high-level language constructs.

2. Background

RISC-V has steadily increased in popularity since its inception in 2011. This interest has come from many players, academic and industry, all the way down to individual hobbyists. Much of this interest is due to the open-source license. However, RISC-V is not a mere pet project by its founders, formal specifications of the instruction set have been ratified by a consortium of participants. This section discusses some of the background of RISC-V, the tools that use the RISC-V instruction set, and the prior art of high-level assembly.

2.1. The RISC-V Instruction Set

The name RISC-V (risk-five) is a nod to the fact that the instruction set is a Reduced Instruction Set Computer, in the same vein as MIPS, Alpha AXP, SPARC, ARM, and many others. These instruction sets favor simple instructions that generally only carry out one task, which can be implemented very efficiently in hardware. However, previous RISC implementations often had the support of a single industry player, and therefore came with restrictions on intellectual property, and required payment of royalties to implement designs of these ISAs. RISC-V, on the other hand, comes with a Creative Commons license that allows royalty-free implementations. The ISA was named RISC-V to highlight that this is the fifth generation of architectures to originate from the University of California at Berkeley.

The RISC-V instruction set definition requires a base integer instruction set in 32, 64, or 128-bits, but permits a variety of optional extensions. This provides for a flexible instruction set that can serve the needs of a wide variety of systems, from embedded systems up to server-class processors. The variations in instruction set are denoted by a naming convention. The base instruction set (for example, in 64-bit registers) is named RV64I. RV64M describes the optional support for hardware multiply and divide instructions. RV64F describes the optional single-precision floating-point instructions. And several other extensions exist for atomic operations, embedded instructions, vector instructions, and so on. The extension provisions also allow individual parties to add custom extensions for which no formal definition has been provided by the RISC-V Foundation.

There is history for open instruction sets. The PISA instruction set [12] was freely available for academic use. A series of tools implementing the PISA instruction set was very popular for research in the late 1990s and early 2000s. However, PISA had limitations that resulted in waning interest. First, the tools required royalties for industry use. Industry support is important for an instruction set to gain major, and sustained, development interest. Other limitations of PISA was that it defined only a non-priviledged

instruction set, and also defined a relatively inefficient instruction encoding (64-bit instructions), intended to allow a high number of custom opcodes for research purposes.

2.2. RISC-V Tools

Because of the industry support and open-source licence, there has been a large number of projects that target the RISC-V instruction set. These tools run the gamut of instruction set emulators [6] [9], processor simulators [7] [11], compilers [1] [7], operating systems [5], and virtual machines [4] [7]. But, because of the royalty-free licensing, hardware vendors can also make processors that implement the RISC-V instruction set. In fact, announcements from Qualcomm, NVIDIA, Microchip Technology, and Google have shown support for developing RISC-V silicon. New companies have even started – SiFive is a fabless semiconductor company who have developed two systems-on-chip (SoCs) that implement the RISC-V instruction set. These SoCs can be purchased as part of two development platforms, the LoFive and the HiFive [2], the former being a minimal embedded system platform and the latter being a more fully-featured system. The BOOM project [13] also provides a hardware implementation of a RISC-V processor, intended to be mapped to an FPGA.

Tools of this nature require a great deal of attention be paid to low-level details. During the development of these tools, and of hardware designs, *dt* can help developers write programs that exercise these low-level details, while still providing a language that allows increased productivity in high-level language features. Thus, *dt* is complementary to existing RISC-V tools, rather than being in competition.

2.3. High-Level Assembly

The *dt* project implements a high-level assembler that targets the RISC-V instruction set. However, there have been past efforts [3] [15] in high-level assembly languages. These past high-level assemblers typically rely on assembler macros to provide some of the same high-level language features that are native to *dt*. The goals of these high-level assemblers is similar to *dt*, but often require that assembly programmers write the high-level language-like macros themselves. Furthermore, none of the existing high-level assemblers target the RISC-V instruction set.

3. Language

This section serves as a brief outline of some of the language features of the *dt* language, and how high-level language constructs produce deterministic machine code output. However, this section does not describe the complete langage definition, interested readers can refer to [16] for a full language reference.

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 statements. A `mem()` block must be supplied with an address, which will serve as the starting address for any statements which will occupy memory in the output file (for instructions or data). *dt* allows for multiple `mem()` blocks to be specified, and will determine the size, in bytes, of each `mem()` block and verify that no addresses overlap and all starting addresses are correctly word-aligned.

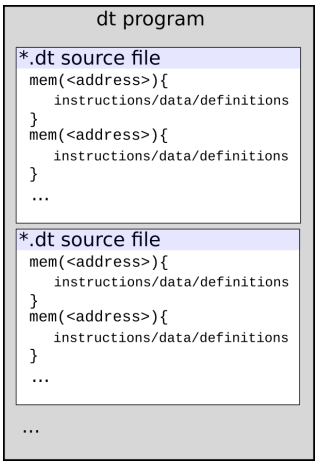

Figure 1. Program organization of a complete *dt* program.

A single *dt* program can be split into multiple files. By convention, those files are named with an extension of `.dt`. There are no restrictions on the number of `mem()` blocks or their locations for a program targeted for the flat memory image output formats. Currently, for the ELF64 Linux executable, there are restrictions that the program entry point address conforms to the Linux ABI.

The statements that appear within a `mem()` block can take one of several forms: comments, name definitions, data values, assembly instructions, and high-level language constructs. The next subsections describe each of these categories.

3.2. Comments and Definitions

Comments can appear anywhere within a *dt* program, both within a `mem()` block, or outside. Comments start with a hash, and continue to the end of a line.

To increase the readability of a *dt* program, names can be defined for memory locations of instructions or data by simply using alphanumeric labels followed by a colon, just like a traditional assembly program. However, *dt* also allows labels to define names for registers. Once a label has been defined for a register, that label name can be used anywhere a register identifier would have been used. Listing 1 shows an example `mem()` block, using these constructs. Register `$x8` is given the name `count` in this example, and is used by an `addi` assembly instruction.

```
1 # This is a comment.
2 mem(0x10000) {
3     # the following line shows a register definition
4     count: $x8
5
6     addi count, $x0, 0 # initialize count to 0
7
8     # ...
9 }
```

Listing 1. Code example of a comment and definitions

Once assembled, this example program will occupy four bytes of memory, for the `addi` instruction only. The register definition does not occupy any memory in the final output. Specifically, the four bytes of the `addi` instruction encoding will occupy addresses `0x10000` through `0x10003`, in little-endian order.

All definitions have global scope, so registers and labelled memory locations can span multiple `mem()` blocks, or even across multiple `.dt` source files.

3.3. Data Values

A programmer is free to put hard-coded literal values into any memory location. Several assembler directives are defined, as shown in Table 1, to allow the programmer to control the amount of memory as well as the format of the initial values.

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"`

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

Data values can be intermixed with instructions, but it is up to the programmer to avoid executing data as an instruction. These directives can be labelled with a definition so that the labels can be referenced by instructions. In the final memory image output, the values that appear after these directives will simply occupy the specified number of bytes. For the `.stringz` directive, there will be one byte per character, plus a null-character that will be inserted at the end of the string. If instructions are listed within the same `mem()` block after a directive, then those instructions will automatically be word-aligned in the output memory image.

Note that it is possible to specify floating-point literals in a *dt* program, but currently there is no support for floating point instructions or registers. The programmer can perform software floating-point, but the intent of these directives is to also add hardware floating-point support in the future.

There is no specific support for setting aside blocks of memory addresses for run-time use of a program. However, if a programmer requires blocks of memory, then the memory can be set aside by using one or more of these directives and simply providing a bogus initial value (for example 0).

3.4. Assembly Instructions

Instructions can be specified by the usual assembly language syntax in the RISC-V ISA specification [8]. In general, this means an instruction mnemonic, followed by register, immediate value, or labelled operands.

The instruction mnemonics permitted are those that are defined in the RV64I base integer instruction set, and the RV64M extension for hardware multiply and divide instructions. The operands permitted will depend on the instruction type. For register operands, *dt* permits registers to be specified using assembler names, for example `$ra` for the return address register, `$sp` for the stack pointer, `$t0` for the first temporary register, and so on. However, numbered registers are also supported using registers `$x0` through `$x31`. Immediate operands are specified using either decimal or hex notation, using the same

syntax as the C programming language.

Control transfer instructions require a target. This target can be specified using labels, where the target instruction has a label definition. In this case, *dt* will calculate the PC-relative offset automatically. Alternatively, the programmer can provide an immediate value, which should be PC-relative offset that will be copied directly into the immediate field of the instruction encoding.

Memory instructions require a register operand (source register for stores, destination register for loads), and a base register and offset operand pair. Square brackets are used to indicate the base register, which should be preceded by an immediate value to indicate the offset.

Many of the pseudo-instructions defined by the RISC-V instruction set are also supported by *dt*. See the technical reference [16] for a table that outline all supported instructions and pseudo-instructions.

```
1  mem (0x10000) {
2      lui $x3, 0x40
3      lw $x5, 0[$x3]
4      beq $x5, $x0, target
5      addi $x5, $x5, 1
6  target:
7      sw $x5, 4[$x3]
8
9  inf: j inf
10 }
11
12 mem (0x40000) {
13     .word 123
14 }
```

Listing 2. Code example of several assembly instructions

Listing 2 gives examples of several types of instructions. This example has two `mem()` blocks, one for instructions, and another for data. The program starts with a load upper-immediate `lui` instruction that forms a base address in register `$x3`. A load word reads memory, and in this case will put the value 123 into the destination register `$x5`. A branch instruction compares the loaded value with the value in `$x0` (the sink register), which in this case will cause a *not-taken* branch, which will execute the fall-through `addi` instruction that increments `$x5`. The store word saves the value of `$x5` to the effective address `0x00040004`. Since there is no way to otherwise halt this program, this *dt* program busy-spins by using an infinite loop using a jump instruction that targets itself.

3.5. High-Level Language Syntax

While writing assembly programs is helpful for debugging low-level tools and simulators, the real strength of *dt* is in its support for high-level language constructs. This section highlights many of these features, and how they produce deterministic assembly output. These high-level constructs can be arbitrarily mixed with assembly syntax, and can also take advantage of named register and memory definitions.

3.5.1. Assignments and Arithmetic Operators. 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), the assignment operator, and one of the operands and operators listed in Table 2. To support predictable output, only a single operation can be done per assignment. Therefore, compound expressions, or operations on three or more operands are not allowed. The reason behind this restriction is that *dt* does not do register allocation nor a run-time stack to support spills and fills. The programmer

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* | *reg*	`or`
reg = *reg* | *imm*	`ori`
reg = *reg* ^*reg*	`xor`
reg = *reg* ^*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 2. Assignment operations allowed by *dt*

must specify the registers or memory locations used to hold intermediate results of what would have otherwise been compound expressions.

Some operations listed in Table 2 require more than one instruction, or require an instruction that may differ from expectation. For example, assigning 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. 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`.

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

Listing 3 shows two `mem()` blocks with equivalent instructions. However, one `mem()` block is written using assembly syntax, and the other is written using the high-level language assignment syntax. These

two `mem()` blocks will produce binary-equivalent instructions. Note that these `mem()` blocks can not appear in the same program, since their memory regions overlap.

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

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

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

Listing 3. Code example of equivalent instructions using (a) assembly syntax and (b) high-level 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 4 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 4. Code example showing uses of the address-of operator

3.5.2. Structured Control Flow. The last group of syntatical constructs provide the high-level language 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 *dt* 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 temporary register. The *dt* compiler also has no formal mechanism or syntatical construct for functions. This is due to the several items that functions require – a run-time stack, the stack pointer, the return register, function arguments, and so on. These requirements are against the intent behind *dt* to give all control to the programmer. If the programmer requires functions, they need to set up a run-time stack manually, and also manually carry out the ABI requirements for arguments, return values, callee vs. caller saved

registers, etc.

The constructs that are available, however, are: `if`-statements, `if-else` statements, `while` loops, `do...while` loops, `until` loops, and `do...until` loops. These structured control flow code bodies can contain instructions, definitions, data values, assignments, and other high-level control flow statements. These code blocks can also be named themselves – simply provide a label definition followed by the entire control flow statement. This will name the first instruction of the code block. The named instruction might be an instruction in the body of the statement (for `do...while` and `do...until` loops), or it might be an instruction that is not evident in the code but is part of the supporting code automatically emitted by the block statement.

***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
	  bne reg, $x0, label9

Table 3. Structured control flow code blocks recognized by *dt*

The condition for each of the statements must be a register or label that corresponds to a register. The meaning, however, is the same as in C – any non-zero value is considered `true`, and zero is considered `false`. Curly brackets are always required, even if the body only requires a single instruction. Table 3 gives the syntax for each of the constructs and the code generated by *dt*. The labels used to support these constructs are intenerally generated by *dt* and are random alphanumeric labels that will not conflict with user-defined labels.

4. Example *dt* Programs

In this section, we highlight two example *dt* programs. These are examples that not only show the syntax of programs, but also show how this tool can be useful in two different ways – for working around a

hardware bug, and for showing the performance of a design. These are real-world examples, based on the past experiences [14] of an author of this paper.

The design being tested was a full hardware design of a dual-core out-of-order execution processor, implemented in the Verilog Hardware Description Language. The processor cores in these hardware designs were *not* implementing the RISC-V instruction set, but the examples highlighted in this section are not ISA-dependent. The design was sent for fabrication twice – first as a lower-cost prototype that could be tested for hardware bugs (of which there were several, despite the months of testing in simulation that occurred before sending the design for fabrication), and the second was the full-cost demonstration [17] of the actual research aspects of the design that incorporated fixes of the hardware bugs found in the first phase.

4.1. Hardware Bug Work-around Example

This example shows how a particularly nasty hardware bug can be avoided with a little bit of clever programming effort. After sending the first phase chip out for fabrication, it was discovered that the simulation of the design had defined a pre-processor token that was including parts of the design in the data cache. But when producing the design files for fabrication, that pre-processor token was not being defined and parts of the data cache were not being included.

The effect of this bug was that whenever a load instruction queried the data cache, the cache would always respond with a cache *hit* – whether the data was actually in the cache or not. If the load should have *missed*, then whatever bogus data was in the cache block would be sent back to the pending load instruction, to be errantly saved to its destination register. However, if the cache block had been previously used by another instruction, then the cache block did have the correct data and execution could correctly proceed with the cached data. This allowed for an opportunity for a work-around. We simply require that all loads have their cache blocks prefetched with the correct data. The performance will always reflect cache hits, and thus give an unfair advantage for performance estimations, but at least the system could demonstrate functional correctness.

If *gcc* was our only tool for creating executables, then it would be nearly impossible to guarantee that all load instructions had their data prefetched. This highlights a prime use-case for *dt*. To prefetch, we simply need to execute load instructions (we'll call them the *prefetching loads*) that use the correct block addresses – these prefetching loads will obtain incorrect values, which we then simply discard. Later, we can use other load instructions (we'll call them the *useful loads*) that will hit with correct values. There are a few other issues that also need careful consideration. First is that before executing useful loads, we need to make sure that all of the prefetch loads have finished populating their cache blocks. This requires a busy-wait loop after all prefetching loads. And second, the working set of data must fit within the data cache. If the working set changes, then additional prefetch loads must be executed to populate the cache with the new working set.

Listing 5 shows a simple *dt* program that sums the odd values of a short array. The array data is incorporated as literal values in a separate `mem()` block – all of which fits in the data cache. A loop is first used to prefetch the array data, followed by a busy-wait loop that ensures all data values have been added to their cache blocks. The last loop carries out the intended program, each iteration of the loop reads an element of the array, determines if it is odd or even valued, and adds the value to the running total if the value was odd.

```
mem (0x10000) {
ii:   $x8                       # give nice names to some of the registers to be used
size: $x9
addr: $x10
cond: $x11
sum:  $x16
val:  $x17

    ii = 0                      # the prefetch loop
    size = 5
    addr = @array
    cond = ii < size
    while (cond) {
        lw $x0, 0[addr]
        addr = addr + 4
        ii = ii + 1
        cond = ii < size
    }

    ii = 0                      # busy-wait loop
    cond = ii < 100
    while (cond) {
        ii = ii + 1
        cond = ii < 100
    }

    sum = 0                     # loop that sums only the odd values of the array
    addr = @array
    ii = 0
    cond = ii < size
    while (cond) {
        lw val, 0[addr]         # read the array element
        cond = val & 0x1        # mask the lowest bit
        if (cond) {             # odd values will have the lowest bit of 1
            sum = sum + val
        }
        addr = addr + 4
        ii = ii + 1
        cond = ii < size
    }

    addr = @result              # save the final result
    sw sum, 0[addr]

inf:                            # infinite loop to end the program
    j inf
}

mem (0x40000) {
array:                          # the array data (5 elements)
    .word 31
    .word 82
    .word 10
    .word 65
    .word 27
result:                         # a memory location to hold the final sum
    .word 0
}
```

Listing 5. Code example showing a hardware bug work-around

4.2. Performance Validation Example

It is also important to be able to show that performance targets are actually met by hardware designs. A second use of *dt* can assist in this goal as well. The aforementioned dual-core processor design that was fabricated in [14] consisted of heterogeneous cores – i.e. two different core microarchitectures. One

core was a 1-wide out-of-order design, and the other was a 2-wide out-of-order design. It was important to show that these cores were able to achieve their peak instruction bandwidth, measured in instructions per cycle (IPC).

The requirement for a processor core to reach its peak IPC is that there are 1) ample instructions, 2) few true data dependencies, and 3) relatively few control transfer instructions. The first requirement is in place to show *sustained* peak IPC. And the other requirements avoid cycles where fewer instructions are completed than the width of the pipeline. If there are a lot of true dependencies, then it is possible to have cycles where there are plenty of instructions that have been fetched, but too few that are ready to execute simultaneously. And if there are a lot of control transfer instructions, then it is possible that bubbles are introduced when trying to fetch instructions in the first place. These requirements are especially important for the 2-wide core.

```
mem (0x00010000) {
ii:   $x8
size: $x9
addr: $x10
cond: $x11
sum:  $x16
val:  $x17

    size = 100000
    sum = 0
    addr = @array
    ii = 0
    cond = ii < size
    while (cond) {              # unrolled loop body
        lw val, 0[addr]
        sum = sum + val
        lw val, 4[addr]
        sum = sum + val
        lw val, 8[addr]
        sum = sum + val
        lw val, 12[addr]
        sum = sum + val

        addr = addr + 16
        ii = ii + 4
        cond = ii < size
    }
}
```

Listing 6. Code example with peak instruction bandwidth

Listing 6 is a modification of the array element-summation program. The prefetching and busy-waiting loops have been omitted for clarity, as well as the `mem()` block of array data. The main loop that sums array elements has been modified to sum *all* array elements – avoiding branch instructions within the loop body to meet the third requirement from the previous paragraph. Also, the number of elements in the array have been substantially increased to meet the first requirement – it's during this loop that we expect to achieve peak IPC. To reduce true data dependencies, we unrolled the loop body to sum four array elements per iteration. Thus, the loads are all independent of each other, and only depend on the address calculation. There is still a true dependence on the loop counter and condition check. If concerns arise that too many true dependencies still exist, then the loop body can be unrolled to sum eight or sixteen elements per iteration.

5. Conclusion and Future Work

The *dt* assembler is still a work in progress, and will continue development for the foreseeable future. The scanner is written in *flex*, and consists of roughly 180 lines of code. The parser is written in *bison*, and is roughly 3,000 lines of code. The assembler writes output to both flat memory image files, and to statically-linked ELF64 Linux executables. Most testing of *dt* thus far has focused correctness of the Linux executables. Those executables are run on an actual development board, the HiFive Unleashed [2] by SiFive, which has a RV64GC quad core SoC capable of running a version of Debian Linux.

There are several features that are slated for future additions to *dt*. Currently *dt* does not support compressed instructions. The compressed instructions are encoded into two bytes, rather than the base instruction encodings that are four bytes. These can be easily added to make more efficient (in size) executables.

Another possibility for future work is to add support for struct-like data types. Structs have a well-defined memory layout, and thus would not counter the goal of *dt* to produce deterministic output. Complex types were not included in this first instance of *dt* simply to confine the scope of project to something manageable for a summer research project. But also, the inclusion of struct-like data types will require careful consideration in the parser.

The output file formats of *dt* provide flexibility to users. The flat file formats can easily be read into processors simulators, emulators, run-time environments, and virtual machines. However, there has been no effort thus far to actually integrate into some of the popular tools [6] [9] [11] being used by the research community. Future work will add additional output file formats that natively match the input format (possibly including checkpoint support) of these popular tools.

The ELF64 output file format could also be modified to add creature comforts. For example, it is possible to add a symbol table and debugging-related segments. This would then allow meaningful symbols to be visible to *gdb* when debugging *dt* programs.

There are also uses of *dt* that could be advantageous in an academic setting. For example, the PI of this project has noticed that undergraduate students occasionally have a disconnect in the distinction between machine language and assembly language. Therefore, a possible use of *dt* in the classroom could highlight this distinction by having assignments wherein students add new syntax to the *dt* grammar, or have students re-implement the code to emit binary instruction encodings.

References

[1] "GNU Toolchain for RISC-V, Including GCC," 2020, Code Repository. [Online]. Available: https://github.com/riscv/riscv-gnu-toolchain

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

[3] "IBM High Level Assembler and Toolkit Feature," 2020, Product Brief. [Online]. Available: https://www.ibm.com/us-en/marketplace/high-level-assembler-and-toolkit-feature

[4] "QEMU - The FAST! Processor Emulator," 2020, Product Brief. [Online]. Available: https://www.qemu.org/

[5] "RISC-V - Debian Wiki," 2020, Reference. [Online]. Available: https://wiki.debian.org/RISC-V

[6] "RISC-V Simulator for x86-64," 2020, Reference. [Online]. Available: https://rv8.io/

[7] "selfie - An Educational Software System of a Tiny Self-compiling C Compiler, a Tiny Self-executing RISC-V emulator, and a Tiny Self-hosting RISC-V Hypervisor," 2020. [Online]. Available: http://selfie.cs.uni-salzburg.at/

[8] "Specifications - RISC-V International," 2020, Manual. [Online]. Available: https://riscv.org/specifications/

[9] "Spike, a RISC-V ISA Simulator," 2020, Code Repository. [Online]. Available: https://github.com/riscv/riscv-isa-sim

[10] "Using the GNU Compiler Collection (GCC)," 2020, Manual. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/

[11] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 Simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, May 2011.

[12] D. Burger and T. Austin, "The Simplescalar Tool Set, Version 2.0," Computer Sciences Department, University of Wisconsin - Madison, Tech. Rep. TR1342, 1997.

[13] C. Celio, "A Highly Productive Implementation of an Out-of-Order Processor Generator," Ph.D. dissertation, University of California Berkeley, December 2018.

[14] E. Forbes, R. B. R. Chowdhury, B. Dwiel, A. Kannepalli, V. Srinivasan, Z. Zhang, R. Widialaksono, T. Belanger, S. Lipa, E. Rotenberg, W. R. Davis, and P. D. Franzon, "Experiences with Two FabScalar-Based Chips," in *Proceedings of the 6th Workshop on Architectural Research Prototyping (held in conjunction with ISCA-42)*, June 2015.

[15] R. Hyde, *The Art of Assembly Language, 2nd ed.* San Francisco, CA: No Starch Press, 2010.

[16] J. Severeid and E. Forbes, "Technical Reference for the *dt* Programming Language and Assembler," Department of Computer Science, University of Wisconsin-La Crosse, Tech. Rep. TR04032020, 2020. [Online]. Available: https://cs.uwlax.edu/~eforbes/dt/

[17] V. Srinivasan, R. B. R. Chowdhury, E. Forbes, R. Widialaksono, Z. Zhang, J. Schabel, S. Ku, S. Lipa, E. Rotenberg, R. Davis, and P. Franzon, "H3 (Heterogeneity in 3D): A Logic-on-logic 3D-stacked Heterogeneous Multi-core Processor," in *Proceedings of the 35th IEEE International Conference on Computer Design*, November 2017, pp. 145–152.