TOT (Terminal-on-TOT) Final Report

1st Stephen Kandeh
Department of Physics and EECS
Massachusetts Institute of Technology
Cambridge, MA, USA
skandeh@mit.edu

2nd Julio Rodriguez
Department of Physics and EECS
Massachusetts Institute of Technology
Cambridge, MA, USA
juliorod@mit.edu

***Abstract*—We present Terminal on TOT (TOT), a machine running on a five-stage 50MHz processor with a 32-bit Instruction Set Architecture (ISA) based on RISC-V. This processor is designed to implement a modified Harvard computer architecture, where two direct-mapped caches are used to mimic a split memory for instructions and data while still referring to a single shared main memory module. TOT comes with a boot-loader stored on a local ROM module which allows him to be programmed by a separate machine through a UART port. Unused address spaces are utilized for MMIO instructions to devices and global processor settings. The necessary infrastructure for exceptions and kernel mode are in currently in place. We implement this design through the Nexys 4 DDR FPGA and evaluated its performance using iVerilog and GTKWave.**

***Index Terms*—Processor, RV32I, Direct-Mapped Cache, DRAM, Distributed RAM, BRAM, UART, RS232, MMIO, Kernel, TOT**

I. Instruction Set Architecture

The Instruction Set Architecture (ISA) used by the processor is custom and loosely based on the RV32I instruction set. [1] An instruction always follows the same format: the upper 6 bits encode the operation of the instruction (opCode), the following 10 bits encode arbitrary registers of 5 bits each (val1 and val2), and the last 16 bits encode either a register or immediate (val 3). A depiction of this is shown in TABLE I.

TABLE I
INSTRUCTION FORMAT

31 — 26	25 — 21	20 — 16	15 — 0
opCode (6 bits)	val1 (5 bits)	val2 (5 bits)	val3 (16 bits)

There are 4 different types of instructions that can be distinguished conceptually:

- 3R Instructions - instructions that involve the use of 3 registers (rd, rs1, and rs2)
- Immediate Instructions - instructions that contain an immediate in val3
- CPC Instructions - instructions that change the PC in some non-trivial way, such as branches or jumps
- Load/Store Instructions - instructions that interact with the main memory of the machine

There are a total of 25 instructions that are recognized by the processor. A reference of these instructions is provided in TABLE II.

TABLE II
INSTRUCTIONS

Instruction	Syntax	Description
ST	ST rs2 rs1 offset	mem[rs1 + offset] $\Leftarrow$ reg[rs2]
LD	LD rd rs1 offset	reg[rd] $\Leftarrow$ mem[rs1 + offset]
ADD	ADD rd rs1 rs2	reg[rd] $\Leftarrow$ reg[rs1] + reg[rs2]
SUB	SUB rd rs1 rs2	reg[rd] $\Leftarrow$ reg[rs1] - reg[rs2]
AND	AND rd rs1 rs2	reg[rd] $\Leftarrow$ reg[rs1] & reg[rs2]
OR	OR rd rs1 rs2	reg[rd] $\Leftarrow$ reg[rs1] $\mid$ reg[rs2]
XOR	XOR rd rs1 rs2	reg[rd] $\Leftarrow$ reg[rs1] $\wedge$ reg[rs2]
SRL	SRL rd rs1 rs2	reg[rd] $\Leftarrow$ reg[rs1] $>>$ reg[rs2]
SRA	SRA rd rs1 rs2	reg[rd] $\Leftarrow$ reg[rs1] $>>>$ reg[rs2]
SL	SL rd rs1 rs2	reg[rd] $\Leftarrow$ reg[rs1] $<<$ reg[rs2]
LUI	LUI rd imm	reg[rd] $\Leftarrow$ imm $<<$ 16
ADDI	ADDI rd rs1 imm	reg[rd] $\Leftarrow$ reg[rs1] + imm
SUBI	SUBI rd rs1 imm	reg[rd] $\Leftarrow$ reg[rs1] - imm
SRLI	SRLI rd rs1 imm	reg[rd] $\Leftarrow$ reg[rs1] $>>$ imm
SRAI	SRAI rd rs1 imm	reg[rd] $\Leftarrow$ reg[rs1] $>>>$ imm
SLI	SLI rd rs1 imm	reg[rd] $\Leftarrow$ reg[rs1] $<<$ imm
JAL	JAL rd label	reg[rd] $\Leftarrow$ pc + 4 pc $\Leftarrow$ pc + label
JALR	JALR rd rs1 offset	reg[rd] $\Leftarrow$ pc + 4 pc $\Leftarrow$ reg[rs1] + offset
BGE	BGE rs1 rs2 label	pc $\Leftarrow$ reg[rs1] $\geq_u$ reg[rs2] ? pc + label : pc + 4
BLT	BLT rs1 rs2 label	pc $\Leftarrow$ reg[rs1] $<_u$ reg[rs2] ? pc + label : pc + 4
SBGE	SBGE rs1 rs2 label	pc $\Leftarrow$ reg[rs1] $\geq_s$ reg[rs2] ? pc + label : pc + 4
SBLT	SBLT rs1 rs2 label	pc $\Leftarrow$ reg[rs1] $<_s$ reg[rs2] ? pc + label : pc + 4
BEQ	BEQ rs1 rs2 label	pc $\Leftarrow$ reg[rs1] $==$ reg[rs2] ? pc + label : pc + 4
NOP	NOP	Ignore instruction
HLT	HLT	Stop the program

Note that val2 is zero for instructions with only 2 arguments.

II. Main Memory and Caches

Tot implements a modified Harvard computer architecture. This means that there is a single memory (DRAM) that holds both the instructions and data for TOT, but two direct-mapped caches – an instruction cache and a data cache – are implemented within him to mimic a split memory. The other two memory structures within TOT are his ROM and MMIO for defining startup behavior and external communication/global setting control.

A. Instruction and Data Caches

To mitigate the slow read/write times of DRAM and maintain a distinction between instructions and data, two direct-

mapped caches are currently implemented. The instruction cache is implemented through Distributed RAM to ensure single-cycle reads upon request hits. The data cache is implemented through BRAM, as we cared less about timing for data-accesses. Both caches are one block wide (each block is one word wide, so each cache has one word per line), and all memory access state machines have clocked state transitions and combinational access signals to ensure no wasted clock cycles.

Every line in the cache is stored through an address. For the data cache, this instruction is formatted in the following way: the lower 2 bits are the offset bits and always zeroed out, the following 9 bits correspond to the cache index, and the upper 21 bits correspond to the tag. This is illustrated in TABLE III. As such, each line in the cache must be 53 bits wide (32 data bits + 21 tag bits). The instruction cache has variable size for optimization purposes, but currently is 64 lines long. This means it has 6 index bits and 26 tag bits. It will also only ever send load requests, so its controlling logic is much more simple than its partner cache. The following information only applies to the data cache.

TABLE III
CACHE ADDRESS FORMAT

31 11	10 2	1 0
tag (21 bits)	index (9 bits)	offset (2 bits)

Two one-hot encoded vectors are kept to indicate each line's validity and cleanliness (dirty lines are written back to main memory upon eviction). A state machine is running to appropriately read and write to BRAM and DRAM depending on the memory access pattern. While running, the entire pipeline is stalled.

B. Main Memory

The main memory of the machine is implemented through DRAM. A request handler routes requests to DRAM, and gives precedence to data cache requests in the case of simultaneous accesses. This is because the data cache state machine will take longer to complete *and* it stalls the entire processor while it makes the request: the instruction cache will simply input NOPs into the pipeline while it makes its request, so parallel work can be completed if we make the instruction cache wait longer.

We are using a memory interface generator IP module to interface with the on-board DRAM resource. Reads and writes are done 16 bytes at a time, so a mask is enabled to ensure only the first 4 bytes of any given request is written to/read from. The cache making the request will stall for a single cycle on writes (once the request is accepted by DRAM) and for as many cycles as necessary to return valid data on reads (typically 20 cycles). No simultaneous reads/writes are accepted (for example, on a load dirty miss, data is first written back, then read from DRAM in two separate requests).

The clock DRAM uses is different from the processor clock, so a BRAM clock crossing bridge is utilized on any signals crossing domains in order to smooth out any meta-stability issues.

C. ROM/MMIO

The address space of DRAM is defined to end at 0x00_FF_FF_FF (although the true space extends slightly beyond that). The MMIO space begins at 0x01_00_00_00 and ends at 0x10_00_00_00. These spaces can be utilized for any communication with the outside world and global processor settings. Each device/setting address comes in pairs: one address for the value intended for the corresponding device/setting, and another address for an indication of fresh data (although this second address won't always be necessary in practice).

There are currently 4 MMIO address: two for UART communication (UART_data and UART_fresh) and another two for cache-bypassing. Our protocol for UART is currently only one way but could be easily modified for two way communication. Our UART protocol is described in more detail in a later section.

ROM memory will contain any important programs that must be performed at startup. At the moment, it only contains a simple boot-loader and described in more detail in a later section. ROM is only accessible from the fetch stage, and for any address above 0x10_00_00_00, requests go directly to ROM as opposed to MMIO, the instruction cache, or DRAM.

III. MODULAR OVERVIEW

TOT is split into three main modules: The processor, DRAM, and peripherals (currently just the UART port). This general structure is illustrated in Fig. 1.

A. Processor Components

a) Fetch: At the Fetch stage, the processor fetches an instruction from the instruction cache whose address is determined by the PC register's current value. This register can be updated in the following ways:

- If the processor is stalling due to a data hazard or an ongoing memory request (in the instruction or data cache), the value in the PC register will remain unchanged until the stall is resolved.
- If a jump instruction is committed in Writeback, the PC register will jump to the indicated address.
- If none of the above conditions are met, the processor updates normally by adding 4 to the PC register.

The 32-bit instruction combinationally passed to decode will be a NOP if there are any stalls, jumps, or if a HLT/NOP instruction is pulled from the cache. Upon a reset, the first instruction is fetched from a user-defined location (currently set to address 4).

b) Decode: Decode accepts a 32-bit instruction for its input register, and combinationally sets 9 control variables for execute:

- **aluOp (5 bits)** - Instructs the ALU what operation to perform on its inputs.
- **rd (5 bits)** - Propagated to WB stage for destination reg.

- **changePC (1 bit)** - Capable of ordering a jump to an appropriately calculated address and flush (defined below) when combined with other control signals.
- **memAccess (1 bit)** - Indicates a LD or ST instruction.
- **wbEnable (1 bit)** - Indicates that a register will be written to.
- **isImm (1 bit)** - Ensures ALU sees a provided immediate number rather than a register output.
- **imm (32 bits)** - The actual immediate to be operated on (only 32 bits to facilitate the LUI immediate function: at most 16 bits if taken from any other immediate instruction)
- **rs1, rs2 (5 bits ea.)** Register addresses sent to an external register file to provide the ALU's

These 9 control signals are capable of uniquely identifying all of our current 25 instructions. rs1 and rs2 reference an external register file module and combinationally receive the data stored at these respective addresses. All of this information is packed into a 110 bit dInst wire and routed to Execute. Under a flush (or reset), Decode will ignore its current instruction and prepare to accept a new one from Fetch. Upon a stall, a NOP decoded instruction is passed to Execute and Decode will hold its current instruction until the stall passes. The pipeline is setup to ensure that a flush and stall never occur simultaneously.

c) Execute: Within Execute, the current decoded instruction is set sequentially from the output of Decode. This register commands the ALU to perform a specified operation, which then outputs the aluOut and branch wires. The first input to the ALU is always a register file output, while the second may be an immediate value. The isImm and changePC control signals utilize branch to distinguish between JAL, JALR, and any branch instructions. The jump and nextPC signals are then appropriately set. An external PC pipeline is run at the top level in a manner that ensures Execute and Memory always get the correct address for their instruction (which is necessary for various calculations). A 103 bit wire called potentialMemoryReq is then combinationally forwarded to Memory containing the control signals memAccess, aluOut, rs2Val, wbEnable, rd, nextPC, and jump.

Under flushes and resets, the current decoded instruction is cleared, and an output ensuring Memory and Writeback do nothing is sent. Under memory stalls, Execute will hold its current decoded instruction and output the same signals (knowing memory will not be listening until the memory stall is lifted). Note that Execute, Memory, and Writeback are not affected by data hazard stalls. They are allowed to clear their instruction through the pipeline precisely to eliminate the hazard stall. The logic for this type of stalling is handled by a separate, external module (called RegConflict).

d) Memory: The decision to split the memory access into its own separate stage was made in the face of impossible timing constraint violations. The input to Memory is received sequentially. If a memory request is not required, the signals are forwarded to Writeback unchanged. Otherwise, a request with the appropriate write, data, and address signals is sent to the data cache. This is a separate module following the protocols briefly outlined in the Main Memory and Caches Section. Memory will raise a memory stall while waiting for the green light from the data cache, and holds onto its current instruction while doing so.

After retrieving a response and determining the appropriate data to write back (if relevant), commands (wbEnable,rd,wbData,nextPC,jump) are combinationally forwarded to Writeback in a 71 bit wide wire.

As with the previous two modules, under flushes and resets, the current commands are cleared.

e) Writeback: This is the stage in which exceptions are committed. When an exception is triggered, kernel registers epc and exc are utilized. Epc will be populated with the program counter of the instruction that triggered the exception in case we must resume the program. Exc will be populated with the address of a pre-written program specialized with handling the particular problem that triggered an exception. At the moment, we have exceptions for illegal memory accesses and opcodes, incorrect privilege, interrupts, and page faults. The next step in dealing with a triggered exception is to jump to a general TRAP address. This is a basic program that adds the user's register state onto the stack. It ends with a jump to whatever address is in exc. Upon returning, the TRAP program proceed to reload the user's state and jump pack to epc to resume the user program.

In addition to dealing with exceptions, Jumps, register values, and PC values are all committed in the writeback stage. It is not affected by any stalls or flushes.

f) Exception/PC Pipeline: These are pipelines running parallel to the processor that propagates the PC and exception value for a given stage to writeback in a way that takes into account the different rates of instruction flow due to memory, hazard, and fetch stalls.

IV. Peripherals/Processor Settings

A. UART Port Communication

The Nexus 4 DDR FPGA includes a UART port which is normally used to upload a build. With this same port, a separate computer can send data to TOT through a Python script and effectively write programs directly into main memory.

The information sent to and received by TOT is formatted in accordance to the RS232 protocol for a baud-rate of 2,000,000 bits/sec and one stop bit. The UART line is active low; a start bit with a value of 0 indicates the start of a transmission, and a stop bit with a value of 1 – succeeding the 8 data bits transmitted in little-endian order – indicates the end of a transmission (no parity bit is transmitted). [2] When a full word is received, it is written to the UART_data MMIO address. A one is placed in UART_fresh to indicate that new data is present. When data is collected by TOT, he must always remember to set UART_fresh low again and await new information.

B. Processor Settings

Currently, the only processor setting is a cache-bypass address. When it is set high, any store request will be sent directly to DRAM. This ensures consistent cache states and allows the user to execute self-modifying code on TOT.

C. Bootloader

TOT includes a hardwired bootloader to handle data incoming from the UART port. This loader is essentially a 3-stage state machine. The first word it expects from the UART port will always be the expected start address of the user program. Then a continuous loop executes in which TOT first receives and stores an address, then an instruction, then stores directly to DRAM. This happens continuously until -1 is sent over UART (since no address or instruction can ever be all F's). At this point, 2 (rather than 1) is stored in the UART_fresh MMIO address, and TOT jumps to the start address. Stack and global pointers are then set, and MMIO protocol is maintained.

V. Retrospective Thoughts

A. Vivado is Hard

It's interesting to realize how fast things moved when they were divorced from Vivado and its GUI. The entire processor was mostly completed within the first weekend/week. Things slowed from a sprint to a crawl after we started the journey of interfacing with DRAM. A week was spent simply becoming familiar with Vivado to an extent where meaningful debugging work could be accomplished.

B. Timing is Hard

Perhaps the most frustrating aspect of this project was trying to navigate the flood of timing errors we got in trying to run TOT at 100MHz. Trying to track down problems quickly became seemingly impossible, and was only worsened by the fact that Vivado refused to use the relevant user-defined name for wires. Instead, wires would be labeled in terms of indices into obscure arrays they originated from. In addition, pushing the compiler to optimally route a board with a high clock frequency made build times go from 5 to 25 minutes. I'd like to return to these issues once TOT is more stable to learn how to deal with them more effectively.

C. Hardware Bugs are Hard

At the beginning of 6.111, I realized how much I missed software bugs. Soon into final projects, I realized how much I missed simulation bugs. By the middle of Tot's timeline, making progress absolutely required working solely on the hardware directly, because MIGs and other IPs that I could not simulate became a huge part of the project. I became quite accustomed with ILA probes, except these quickly became problematic after introducing UART, since the timescales of UART of much larger than processor clock time scales.

D. Inconsistent Bugs are Hard

I've seen more seemingly random and disconnected bugs in TOT's hardware development than ore than any previous project but we're running out of time so that's all I'll say for now

VI. References

References

[1] SiFive Inc. Andrew Waterman, Krste Asanovi. *The RISC-V Instruction Set Manual Volume I: User-Level ISA*. CS Division, EECS Department, University of California, Berkeley, May 2017.

[2] National Instruments. *Serial Quick Reference Guide*, June 2018.

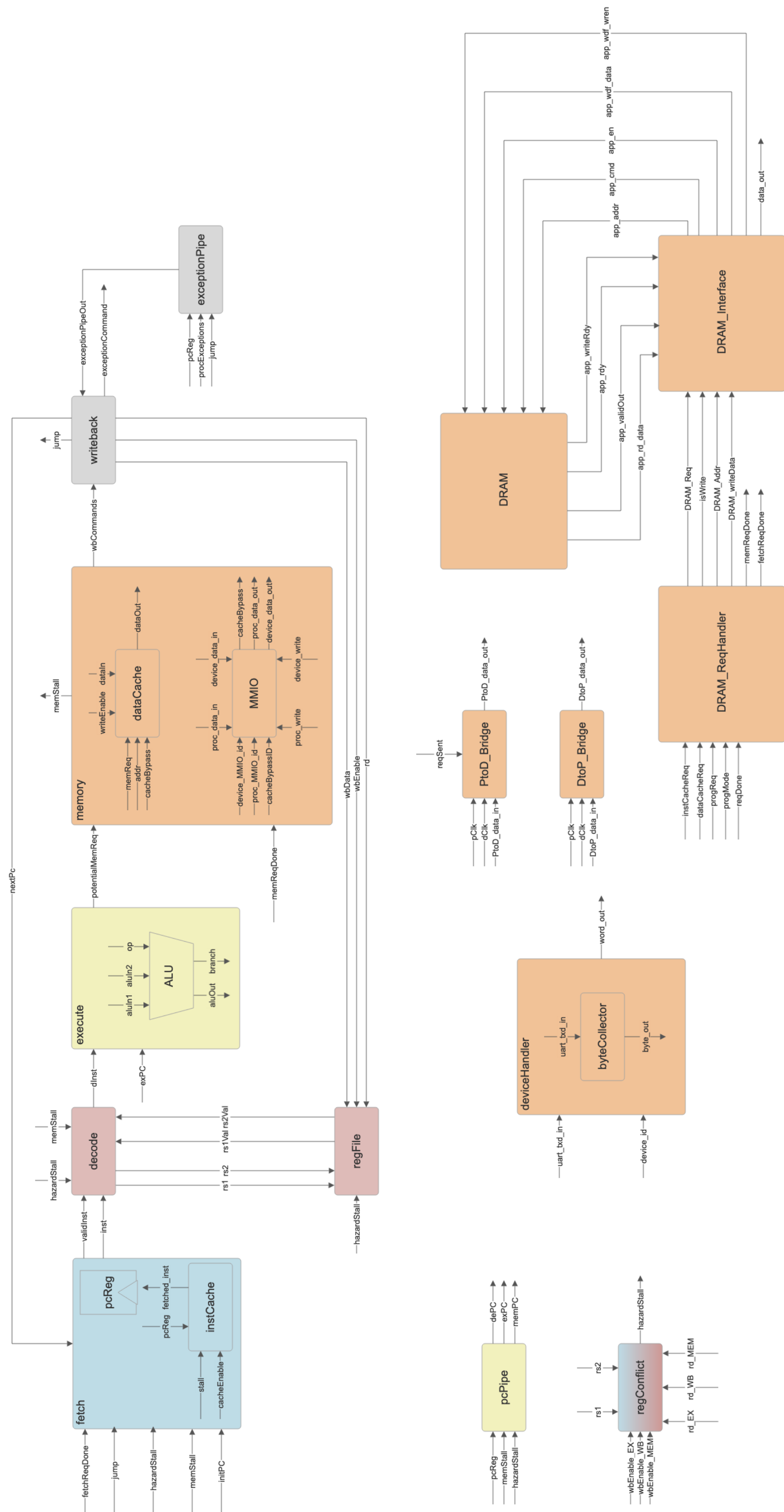

Fig. 1. Complete block diagram of TOT. The diagram is color-coded to highlight the 5 pipelined stages of the processor. Note that the DRAM and peripherals modules are handled within the memory stage of the processor.