

The μ Toad Proof of Concept

Jacob Nelson

March 18, 2004

Chapter 1

Introduction

The μ Toad Proof of Concept (μ Tpoc) is a simple microcontroller that is for the most part compatible with the basic PDP-6/PDP-10 architecture. It exists to demonstrate the feasibility of the μ Toad concept.

The μ Toad is a simple microcontroller that is for the most part compatible with the basic PDP-6/PDP-10 architecture. It is intended to be able to be used as an in-house embedded microprocessor for a variety of control applications.

XKL has many years of experience with the PDP-10 architecture. It is beneficial to use existing development tools and computers to write and debug software that will run on the μ Toad. We can write code on a real Toad, debug the code on the real Toad, and then run the same code on an embedded μ Toad in a chip.

XKL has a need for a simple microprocessor to be used as a control element in various designs. The first is the fan controller. The μ Toad was designed first and foremost to work as a fan controller, and then to be flexible for other purposes.

The μ Toad is *not* intended to be a multiuser computer that runs Tops-20. It is intended to be a simple microcontroller for embedded control tasks. A development path can certainly be followed from the μ Tpoc to a multiuser Tops-20 machine, but different design decisions will be made for that development path than will be made for the microcontroller development path.

The μ Tpoc is almost, but not quite, a design ready for the fan controller. A few more instructions and modifications from lessons learned in the lab will make for a flexible μ Toad for the fan controller.

This document describes what we planned for the μ Toad architecture, what we did to implement the μ Tpoc, and where we would like to go with the μ Toad concept in the future.

Chapter 2

Architecture and Implementation

The μ Tpoc is intended to execute a subset of the PDP-10 instruction set, so that binaries can be created that run on both a real Toad and a μ Toad. The μ Toad is intended to be more like an embedded microcontroller than like its multiuser mainframe PDP-10 relatives. The architecture was designed to work well in embedded control applications.

2.1 Instructions

The instruction set chosen for the μ Tpoc was intended to be simple, fast, and easy to implement. It was intended to be complete enough to write useful software for the fan controller, but simple enough to be easy to implement and easy to debug.

2.1.1 Supported instructions

All instructions behave as they would on a single-section (section 0) PDP-6 or PDP-10.

Move instructions (200-217) Supported, currently without flags.

EXCH (250) Supported.

AOBJP/AOBN (252-253) Supported.

JRST (254) Two forms of JRST are currently supported. The HALT instruction, which is a JRST where the AC field is 4, halts the machine. All other AC field specifications for a JRST currently behave like a JRST 0.

JSP (265) Supported.

Add/Subtract (270-277) Supported, currently without flags.

Compare/Jump/Skip instructions (300-377) Supported, currently without flags.

Logic instructions (400-477) Supported.

Halfword instructions (500-577) Supported.

Test instructions (600-677) Supported.

μ Toad IO instructions (770-775) The new μ Toad-specific IO instructions are supported, as described in section 2.3.

2.1.2 Unsupported instructions

Since the μ Toad executes only a subset of the PDP-10 instruction set, there are many unsupported instructions. Some of these would be easy to implement, and some would be difficult. Any instruction listed below is regarded as illegal, and halts the processor with an illegal opcode error. If UWO handling were enabled, control would be passed to the UWO handler rather than halting the processor.

Floating point instructions Not currently supported.

Doubleword instructions Not currently supported.

Byte instructions Not currently supported.

Multiply and divide Not currently supported.

Shift and rotate Not currently supported.

JFBO Not currently supported.

BLT Not currently supported.

JFCL Not currently supported.

XCT Not currently supported.

Stack instructions Not currently supported.

JSR Not currently supported.

JSA and JRA Not currently supported.

APR instructions The XKL-1 APR instructions are not currently supported.

XKL-1 special moves The XKL-1 PMOVE, NMOVE, AMOVE, and UMOVE instructions are not currently supported.

All others All instructions not listed in section 2.1.1 are not currently supported.

2.2 Memory

The μ Tpoc has an 18-bit address space. It uses the old-style (section 0) effective addressing scheme, with the difference that indirect addressing is not currently supported. If the X field of the instruction word is nonzero, the Y field added to the contents of the AC addressed by the X field gives the effective address. If the X field is zero, the Y field gives the effective address directly.

If the indirect bit is set in the instruction word being executed, the μ Toad will halt with an illegal instruction error.

The μ Tpoc is designed with separate instruction and data memory ports. While these connect to a dual-port memory in the μ Tpoc, they could just as easily connect to two separate memories. One could build a μ Toad with a 1K word instruction space and a 4K word data space, or a 2K word on-chip instruction space and a 256K word off-chip data space. Note that this eliminates the possibility of self-modifying code. Any instructions written into the data space cannot be executed.

Support also exists for memories that return data after more than one clock, but it is currently not used.

2.3 Input/Output

The μ Tpoc has a very simple IO scheme, intended to satisfy the needs of the fan controller and other simple microcontroller-like applications of the μ Toad. This scheme is similar to, but different from, any other PDP-6 or PDP-10 scheme.

There are 16 IO buses. Each bus has two ports: a control port and a data port. The widths are configurable from 0 to 36 bits. In the current implementation, each port is 16 bits. The control and data ports' natural sizes, however, are 18 bits and 36 bits, respectively. The ports are simply registers. Writing a value to the control port simply sets bits in the control output register. Reading bits from a data port simply copies the contents of the data input register (which is always capturing data) to memory. Peripherals may run synchronously or asynchronously with the processor, as long as metastability is taken into account. IO output registers are set during the store execution cycle.

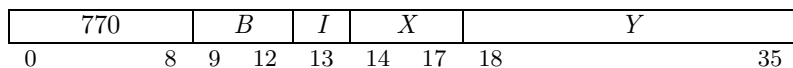
Figure 2.3 shows the ports on one IO bus.

To aid in debugging, the control outputs of IO bus 0 are looped directly to the control inputs. Likewise, the data outputs are looped to the data inputs.

We have defined six IO instructions, as follows. The symbol E refers to the effective address.

CIN

Control Input



Copy data from the control port on the bus given by the B field to the memory location specified by E . If the control port is smaller than 36 bits, the

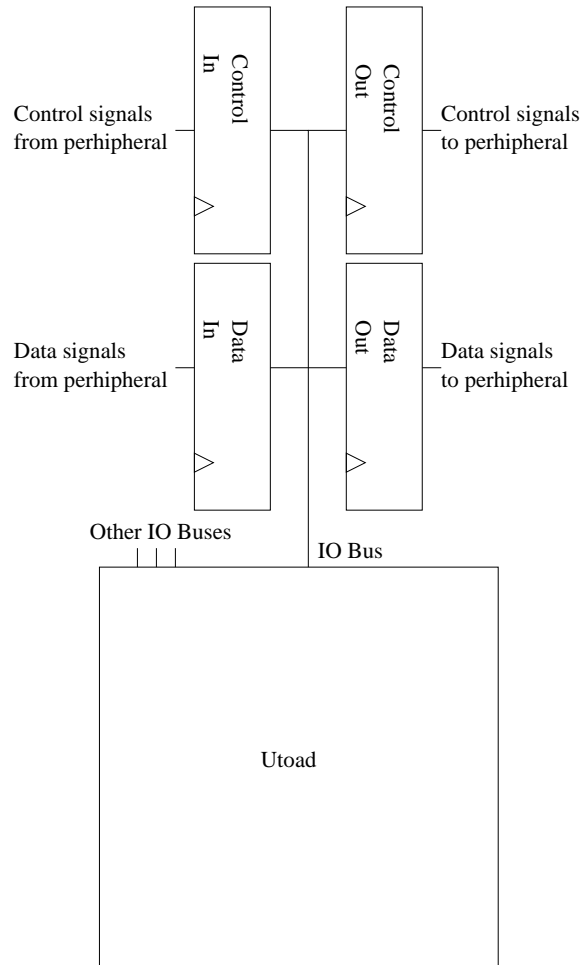
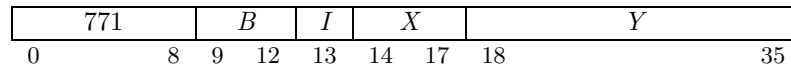


Figure 2.1: μ Toad IO port arrangement

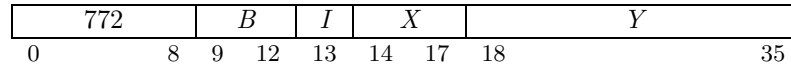
input data is right-justified, and the unused bits in the destination word are cleared to 0.

COUT **Control Output**



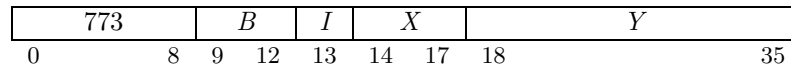
Set the control port on the bus given by the *B* field to the immediate value given by *E*. Note that this instruction always uses *E* as an immediate value, and never as a memory address. If the control port is smaller than 18 bits, the unused bits are ignored. If the control port is larger than 18 bits, the extra bits are cleared to 0.

DIN **Data Input**



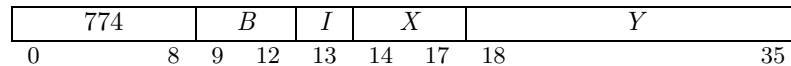
Copy data from the data port on the bus specified by the *B* field to the memory location specified by *E*. If the data port is smaller than 36 bits, the input data is right-justified, and the unused bits of the destination word are cleared to 0.

DOUT **Data Output**



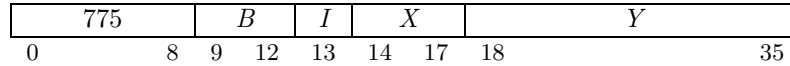
Set the data port on the bus given by the *B* field from the data in the memory location addressed by *E*. Note that this instruction always uses *E* as a memory address, and never as an immediate value. If the data port is smaller than 36 bits, the unused bits are ignored.

CINSZ **Control In and Skip if Zero**



Compare up to 18 bits of the control port on the bus given by the *B* field with the immediate mask given by *E*. If all of the masked bits are zero, skip. If any masked bit is one, continue with the normal execution flow.

CINSO Control In and Skip if One



Compare up to 18 bits of the control port on the bus given by the *B* field with the immediate mask given by *E*. If any of the masked bits are one, skip. If all the masked bits are zero, continue with the normal execution flow.

2.4 Console

The μ Tpoc supports a serial console. The console is implemented by connecting a UART as an IO device. Since the the board we used to implement the μ Tpoc has two serial ports, we implemented two UART's. These show up on the first and second IO buses. The port definitions are the same for both buses. In a different design, we might want to reassign IO bus numbers.

2.4.1 Data port bit assignments

For each UART, the rightmost 8 bits (bits 28–35) of the data register are the character input and output. The rest of the bits (bits 0–27) are unused and reserved.

2.4.2 Control port bit assignments

The control register has the following bit assignments:

Bits 0–23 Unused and reserved.

Bit 24 TX write enable to UART. Pulse high to transmit a character.

Bit 25 RX read enable to UART. Pulse high to discard one character from the input buffer.

Bit 26 TX ready from UART. When high, the UART is ready to transmit a character.

Bit 27 RX buffer not empty from UART. When high, there is a character waiting to be read at the data in port.

Bits 28–35 Received character. Same as data in bits 28–35.

All control bits writable by a COUT may be read back in the same location by a CIN.

See section 3.5 for examples of UART interface code.

2.5 Traps, Interrupts, and UUO's

Note: Code to support interrupts and UUO's currently exists in the μTroc , but it is not enabled.

Interrupts are supported through two interrupt signals coming into the μTroc . One is a special console interrupt, which is intended to start a special console IO handler. The other is a generic IO interrupt, which would start the IO interrupt handler.

I created a simple interrupt controller, which operates as a peripheral to the μToad . It accepts one interrupt line from each IO bus, masks them with a mask register writable via IO instructions, and issues a single interrupt line to the CPU. Since interrupts are currently not used, it is also not used. A more complex controller could be designed.

In the μTroc , any illegal opcode is interpreted as a UUO. If the UUO facility were enabled, illegal opcodes would trap to a UUO handler. Since the facility is not currently enabled, the machine currently halts with an illegal instruction indication upon executing an illegal opcode.

Since there are currently no flags, there are currently no traps.

2.6 Instruction Execution

Since the μTroc is intended to be used as a fan controller, speed is unimportant. We set a goal of a 100 ns instruction cycle, and we achieved that goal. Faster is, however, better.

Instructions are currently executed in three cycles:

Decode/EAcac In the first stage, we begin decoding the instruction and calculating the effective address. The register specified by the *AC* field is read and forwarded to the next stage. The register specified by the *I* field is read and added with the contents of the *I* field to obtain the effective address. This value is presented to the data memory to be read on the next clock and also forwarded to the next stage for use as an immediate.

Execute In the second stage, we perform an operation on the data from the data memory, the AC's, and/or the effective address calculation. Each functional unit decides whether it jumps, skips, writes memory, or writes an AC. These results, along with memory and AC data to be written, is forwarded to the next stage.

Store/Fetch In the third stage, the jump and skip signals sent from each functional unit are used to determine the next PC, which is presented to the instruction memory to be fetched at the next clock. The write enable signals and data from each unit are condensed into write enables and data presented to the register file and the data memory, to be written at the next clock.

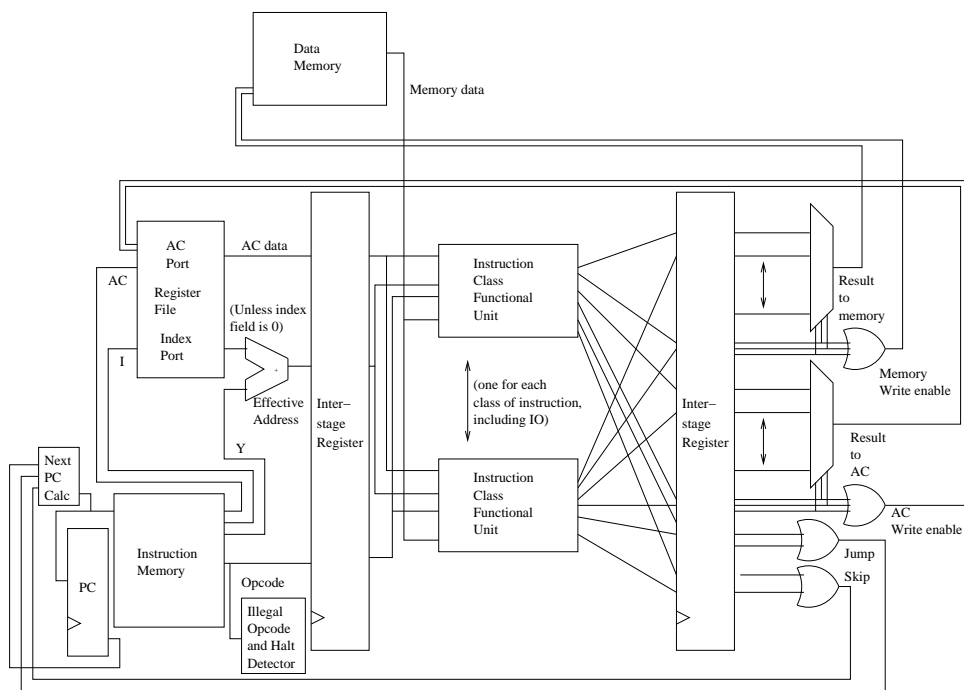


Figure 2.2: Basic block diagram of μTroc .

Figure 2.6 contains a block diagram of the datapath.

The design is not pipelined, but it is very close to being pipelined. Currently, a shift register causes the PC not to advance until the current instruction has gone through all three stages. Turning all the shift register bits on, checking that all results came from the correct registers, and adding code to do hazard detection and data forwarding would finish the pipelining.

2.7 Implementation results

The path from the data memory through the execution stage limits the clock speed of the μ Tpoc. Currently, with 512 words of shared instruction and data memory, we can run at 66 MHz in a Virtex-II 6000 (-4 speed grade). This means that each instruction takes three 15 ns cycles, for a total instruction cycle time of 45 ns. With 32K words of memory, the clock rate falls to something between 44 MHz and 50 MHz, but we run it at 33 MHz for quick PAR's. I have not done any optimization for speed in the design, so it is likely that we can improve the cycle time further.

The design currently uses 925 out of 67,584 slice flip flops in the Virtex-II 6000, which is one percent of those available. It currently uses 2,493 out of 67,584 LUT's, or three percent of those available. Overall, we use 1,814 out of 33,792 slices, for a total slice utilization of five percent. The 32K of RAM takes up 64 block RAM's.

2.8 Code organization

My toplevel design file is called `utoad.vhd`. It instantiates and connects together a DCM, the μ Tpoc datapath, a memory, two UARTS, and the physical IO pins.

The memory (`memory.vhd`) simply instantiates a coregen memory (contained in the subdirectory `coregen_memory`).

The UART module (`uart.vhd`) instantiates a uart receiver (`uart_receiver.vhd`) and a uart transmitter (`uart_transmitter.vhd`), and contains some control logic to interface between them and the μ Toad.

The datapath (`datapath.vhd`) contains the bulk of the design. It instantiates all the functional units for the different classes of instructions, and it contains the program counter, the register file, the IO registers, the debuggers (from `debugger.vhd`), and everything else.

Each class of instruction has its own VHDL component. These files include `moves.vhd`, `aobjs.vhd`, `jrsts.vhd`, `jsps.vhd`, `addsubs.vhd`, `compares.vhd`, `jumpskips.vhd`, `logics.vhd`, `halfwords.vhd`, `tests.vhd`, `exchs.vhd`, and `ios.vhd`.

There is a file called `types.vhd` which contains type definitions to make the code more readable. Almost all files reference `types.vhd` Files which have something to do with IO reference a file called `io_types.vhd`, which contains type definitions for IO. The datapath also references a file called `addresses.vhd`, which contains addresses for the console, interrupt, and UO handlers.

2.9 Debugging

2.9.1 Software

John has written a simple DDT clone that lives in the addresses from 140 to 777. When the machine is reset, the debugger is started. (This would be changed in a production implementation.) The debugger can examine and deposit data in registers and memory, start execution at an arbitrary location, and download code from a real Toad over the serial line. It does not currently support setting breakpoints, although this can be done by hand.

2.9.2 Hardware

The μ Toad includes three instantiations of the trace buffer/debugger described in Appendix B. There is an instantiation for each phase of the instruction execution cycle. The error detect bits of each debugger are connected to the illegal instruction and halt detection circuitry, so the debuggers stop capturing at an illegal instruction or a halt instruction. The logic analyzer can then be used to review the execution of the last 16 instructions. If an illegal instruction has not been detected, the logic analyzer can follow execution of the μ Toad delayed by one cycle in the debugger.

A three-bit signal is brought out to pins to show which part of the instruction cycle the μ Toad is in. The outputs of the three trace buffers are muxed into 36 bits of data output. Bits are assigned for each phase as follows:

Decode/EALcalc The left 18 bits show the current PC, and the right 18 bits show the current calculated effective address.

Execute Various condition bits are assigned as follows:

```
control_debug_in <= "000" &          -- bits 0 to 2

                                '0' &          -- bit 3
                                console_interrupt & -- bit 4
                                uuo_detected &      -- bit 5

                                io_interrupt &     -- bit 6
                                ea_in_ac_r &      -- bit 7
                                halt &           -- bit 8

                                illegal_indirection & -- bit 9
                                illegal_dual_register_write & -- bit 10
                                illegal_opcode &    -- bit 11

                                compare_skips &    -- bit 12
                                jumpskip_skips &   -- bit 13
                                test_skips &      -- bit 14
```

```

io_skips &          -- bit 15
jrst_jumps &       -- bit 16
jsp_jumps &        -- bit 17

aobj_jumps &       -- bit 18
jumpskip_jumps &  -- bit 19
move_writes_ac &  -- bit 20

exch_writes_ac &  -- bit 21
aobj_writes_ac &  -- bit 22
jsp_writes_ac &   -- bit 23

addsub_writes_ac &  -- bit 24
jumpskip_writes_ac &  -- bit 25
logic_writes_ac &   -- bit 26

halfword_writes_ac &  -- bit 27
test_writes_ac &     -- bit 28
move_writes_memory &  -- bit 29

exch_writes_memory &  -- bit 30
addsub_writes_memory &  -- bit 31
jumpskip_writes_memory &  -- bit 32

logic_writes_memory &  -- bit 33
halfword_writes_memory &  -- bit 34
io_writes_memory;      -- bit 35

```

Store/Fetch All 36 bits show the result destined for the data memory if we are not writing an AC or if we are writing an AC because *E* points to an AC. Otherwise, the port shows the result destined for the AC operand.

Other bits are present to indicate illegal instructions or halts.

2.10 Testing

We tested the μ Toad with a modified version of the KL-10 diagnostic from the DEC KLAD pack. John removed (commented out) tests for instructions and features that we didn't implement. John and I wrote code to test some of our new features. When we find bugs, we add tests.

Chapter 3

Using the μ Tpoc

3.1 Instantiating the μ Tpoc

In a real application, we would have a `utoad` wrapper component inside the toplevel design the μ Toad was embedded in. In the μ Tpoc, the `utoad` component is the toplevel design.

This is how I instantiated the μ Tpoc datapath in my μ Tpoc toplevel design file.

```
the_datapath : datapath
  generic map (
    starting_address => 0"000040")    -- start at console handler
  port map (

    -- instruction memory
    instruction_address => instruction_address, -- to memory
    instruction_in      => instruction_in,      -- data from memory
    instruction_read    => instruction_read,    -- read enable
    instruction_ready   => instruction_ready,   -- read data valid

    -- data memory
    data_address        => data_address,       -- to memory
    data_in             => data_in,           -- data from memory
    data_read           => data_read,         -- read enable
    data_out            => data_out,          -- data to memory
    data_write          => data_write,        -- write enable
    data_ready          => data_ready,        -- read data valid

    console_interrupt_in => '0',             -- currently disabled
    io_interrupt_in      => '0',             -- currently disabled
    input_ports_in       => input_ports,     -- array of input ports
    output_ports_out     => output_ports,    -- array of output ports
```

```

-- testbench debug signals
error_opcode_out    => error_opcode_out,
error_address_out   => error_address_out,

-- error detect bits
error_out           => error_detected,
halt_out            => halt_detected,

-- debugger read interface
debug_clk => clk,
debug_data_out => debug_data_out,
debug_error_out => debug_error_out,
debug_cycle_out => debug_cycle_out,

-- main clock and reset lines
clk                => clk,
reset              => datapath_reset);

```

The instruction and data memory ports connect to a dual-port block RAM. The error_out and halt_out signals connect to LED's and the logic analyzer. The debug signals connect to a logic analyzer. The clock signal is from a DCM, and the reset signal comes from a reset tree register.

The IO port signals are arrays of records of ports. Each bus gets an input record and an output record. Each record contains a control port and a data port of configurable widths.

3.2 Connecting an IO device

I implemented the IO ports as an array of records of signals. Here are the type definitions from io_types.vhd:

```

type input_port_type is
  record
    control : std_logic_vector(0 to io_control_width-1);
    data    : std_logic_vector(0 to io_data_width-1);
  end record input_port_type;

type output_port_type is
  record
    control : std_logic_vector(0 to io_control_width-1);
    data    : std_logic_vector(0 to io_data_width-1);
  end record output_port_type;

type input_port_array is array (0 to number_of_io_ports-1) of input_port_type;
type output_port_array is array (0 to number_of_io_ports-1) of output_port_type;

```

The UART on IO bus one is connected in the μ Tpoc toplevel as follows:

```

-- connect the first UART to port 1
first_uart: uart
  generic map (
    FIRST_BAUD_DIVISOR => FIRST_BAUD_DIVISOR,
    SECOND_BAUD_DIVISOR => SECOND_BAUD_DIVISOR)
  port map (
    to_cpu    => input_ports(1),
    from_cpu => output_ports(1),
    console_tx => console_a_tx,
    console_rx => console_a_rx,
    clk       => clk,
    reset     => datapath_reset);
\begin{verbatim}

```

The individual IO ports are broken out inside the UART module as follows:

```

{\small \begin{verbatim}
-- this is for 16-bit IO ports
to_cpu.control <= "0000" &
    txbufwe &          -- TX buffer write enable loopback
    rxread &          -- RX buffer read loopback
    txready &         -- TX ready to transmit
    not_empty_buf &   -- RX buffer is not empty
    rxdata_out;       -- character from receiver
to_cpu.data <= "00000000" & rxdata_out; -- character from receiver
txbufwe <= from_cpu.control(4); -- TX buffer write enable
rxread <= from_cpu.control(5); -- RX buffer read
txbufin <= from_cpu.data(8 to 15); -- Data

```

Rising edge detectors on the txbufwe line and the rxread line keep more than one character from being read or written per pulse.

3.3 Notes on programming the μ Tpoc

3.3.1 Calling subroutines

Since we have not yet implemented the stack instructions, PUSHJ and POPJ are not available. The only instruction that can be used to call a subroutine currently is JSP. JSP jumps and saves a PC word in the specified AC. The word contains the address of the instruction after the JSP in the right half. The left half would contain the current flags, if they were implemented. Since they aren't, zeros are stored in the left half.

When a program calls subroutines with JSP, an AC must be kept free to store the return PC word. If multiple levels of subroutines are desired, the AC with the return PC word must be copied to memory.

It is possible to implement a stack by writing PUSH and POP macros using JSP and other instructions. However, we hope to implement JSR and the stack instructions in the future to make calling subroutines simpler.

3.4 Compiling the IO instructions

To use the μ Toad IO instructions, it is necessary to define the IO opcodes as follows:

```
OPDEF  CIN      [770B8]          ; DEFINE IO INSTRUCTIONS
OPDEF  COUT     [771B8]
OPDEF  DIN      [772B8]
OPDEF  DOUT     [773B8]
OPDEF  CINSZ    [774B8]
OPDEF  CINSO    [775B8]
```

3.5 Using the console UART

Transmitter

The UART transmitter can send one character at a time. To transmit the character "X" on the UART on bus 1, the following Macro code would be executed:

```
CINSO  1,1B26      ; is it ok to transmit a character?
JRST   .-1         ; not yet
DOUT   1,["X"]     ; write character to uart
COUT   1,1B24      ; set the write enable bit
COUT   1,0         ; clear the write enable bit
```

Receiver

The receiver has a 16-byte input buffer. The character at the front of the buffer is displayed at both the control input and data input ports for the bus, along with a bit indicating that the buffer is not empty. After reading a character, the character should be removed from the UART's input buffer by pulsing the RX read line high. A pulse of any length removes only one character from the buffer. To receive a character from the UART on bus 1 into memory location 1234, the following macro code would be executed:

```
CINSO  1,1B27      ; is there data to be read?
JRST   .-1         ; not yet.
DIN    1,1234      ; get character
COUT   1,1B25      ; remove the character from UART receive buffer
COUT   1,0         ; clear the bit. only one character was removed.
```

3.6 Compiling and executing code

Code should be compiled as a standard .EXE file. User programs should start at location 1000, since the space before that is currently used for the microdebugger.

3.6.1 Downloading programs to the μ Toad

User programs can be downloaded to the chip over the serial line. The .EXE file should be run through John's EXEDDT utility to make a .DDT file. The .DDT file is then sent to the μ Toad's console using the TCPF utility on a Toad. To download the file FAN.DDT to our demo board in the lab, we would say "tcpf kent 2025 < fan.ddt".

When a download starts, few characters appear on the screen (a Z and a , maybe). When the download is complete, the last location in the program is printed to the screen. A checksum is run on the program, and if it doesn't match the checksum EXEDDT calculated, three question marks are printed.

3.6.2 Compiling programs into the μ Toad

To compile code into the .bit file for the μ Toad., the .EXE file should be FTP'd to the machine with the μ Toad VHDL sources in binary mode (*not* TENEX!). Appropriate options for width, depth, and output type should be set in memory_maker.vhd, and then the command "make make_memory" should be executed. This will produce an initialization file for a coregen memory model (among other formats). The core should be regenerated with the new initialization file and the design repaired with the new .edn file. This .bit file can then be loaded in the chip and the program executed.

3.7 Debugging code

The microdebugger, called UDT, is small but useful. Here is a list of commands:

<address>\$G Start execution at the specified address.

\$P Proceed execution from a breakpoint or console interrupt. This command doesn't do anything right now, since we haven't implemented a way to get into DDT from a user program.

<address>/ Print the contents of the specified address and wait for a new value to store there.

<control-J> Open and display the next location in memory.

^ Open and display the previous location in memory.

<value> Typing a number and pressing return stores it in the current memory location.

<value>,,<value> Double commas work to separate 18-bit halfwords, just as in a Toad DDT.

<opcode> <ac>,<address>(<index>) Assemble the given numeric instruction into a word. Typing 200 2,1000(7) (a MOVE from location 1000 indexed by AC 7 to AC 2) and pressing enter stores the value 200407001000.

Z Zero memory. This clears locations 1000 and up.

~ Clear checksum register and turn off echo. This command is used by EXEDDT.

| Store data and open the next location in memory turning on echo. This command is used by EXEDDT.

Resetting the machine with the reset jumper causes UDT to start, but the memory is not cleared. Until we implement a console interrupt, this is the only way to get into DDT when running a program.

Chapter 4

Future Directions

We have many plans for the future. We hope to continue to develop the μ Toad to make it faster and more versatile.

We expect to follow two development paths. One will continue to develop a simple embeddable microcontroller. The other will be develop a more complex processor with the intent of running Tops-20.

Here is an unordered list of ideas we have for the future.

- Implement stack operations
- Implement LSH, ROT, and maybe ASH. The C compiler currently uses only LSH.
- Implement IMULI and perhaps more. John says he needs only IMULI and a version of IMUL with an 18-bit memory operand.
- Develop an ethernet interface for the μ Toad.
- To increase our clock speed (and to avoid slowing it down when we add lots of memory), we can add more stages to the execution cycle.
- Spend some time speed-optimizing the design (by moving logic between stages, sharing and/or duplicating logic, etc.).
- Multithread the design to gain parallelism, throughput, and immunity to memory latency.
- Pipeline the design. It is close to being pipelined already; we would need to add hazard detection and possibly forwarding circuitry.
- Fetch multiple instructions at once from a wide memory to do more than one effective address calculation in parallel.
- Develop a real-Toad-based DDT that connects to our micro-DDT through the console so that we can set breakpoints, use the .EXE's symbol table, and do other programmer-friendly things.

- Possibly implement some byte operations. John would like a subset of current byte operations, and it might not be so bad to do in hardware.
- Implement indirect addressing.
- Develop a simplified extended addressing scheme for microcontroller applications requiring more than one section of memory. One possibility is to use 36 bits of the index register when doing indexed addressing, as opposed to just 18.
- Figure out a way so Macro will complain when you use an opcode not supported by the μ Toad.
- Connect the debugger outputs to a user JTAG interface, so they can be read out without a logic analyzer.
- Connect the debugger outputs to an IO port on the μ Toad, so they can be read out by software.
- Add writable trigger words to the debuggers, so they can act as simple logic analyzers.

Appendix A

Tops-20 .EXE format

To execute programs written on a real Toad on a μ Toad, we need a way to load the programs into the μ Toad's memory. To accomplish this, I wrote a memory model in VHDL that parses and loads a Tops-20 .EXE file. This section contains notes I made while understanding the .EXE format.

Tops-20 files are organized in pages of 512 36-bit words each. The first page of a Tops-20 .EXE file contains a directory of the file's contents.

The directory is usually organized in multiple sections. A section is delimited by an IOWD-like word, which contains a section type in the left half and a count in the right half. The count includes the section delimiter word. I know of four types of section delimiter words:

1776,,count Page maps. Each entry in this section is made up of two words that convey information, including:

DEFSTR(EXADR, ,35,27)	;FILE PAGE ADDRESS
DEFSTR(EXSEC, ,26,18)	;SECTION NUMBER (I.E. WHICH 256K ADDRESS SPACE)
DEFSTR(EXCNT, ,8,9)	;COUNT FIELD
DEFSTR(EXPAG, ,35,27)	;PROCESS PAGE ADDRESS

These fields (and more) are split over the two words as shown in figure A.

1775,,count Entry vector

1774,,count PDV address

1777,,count End of directory (count should be 1)

Access bits	File page address
Copy count	Memory page address
0	8 9 35

Figure A.1: Page map.

```
FILDDT>get sys:monitr.exe/d
[Looking at file TOED:<SYSTEM>MONITR.EXE.103052]
```

```
0[ 1776,,45
1[ 100000,,1
2[ 120000,,1000
3[ 100000,,122
4[ 2000,,1121
5[ 140000,,0
6[ 4000,,1124
7[ 100000,,125
10[ 4000,,1131
11[ 140000,,0
12[ 116000,,1142
13[ 100000,,132
14[ 201000,,1433
15[ 100000,,334
16[ 5001
17[ 100000,,335
20[ 14000,,5002
21[ 100000,,352
22[ 3000,,5017
23[ 100000,,356
24[ 227000,,5023
25[ 100000,,606
26[ 22000,,5755
27[ 100000,,631
30[ 6433
31[ 100000,,632
32[ 31000,,6434
33[ 140000,,0
34[ 17000,,6466
35[ 100000,,664
36[ 64000,,6506
37[ 140000,,0
40[ 3000,,6573
41[ 100000,,751
42[ 16000,,6577
43[ 140000,,0
44[ 6616
45[ 1775,,3
46[ 5
47[ 1,,161
50[ 1774,,2
51[ 1,,133330
52[ 1777,,1
53[ 0
```

Figure A.2: Dump of the first few words of a Tops-20 monitor.

```
[toed] DXX:<7.EXEC>! get sys:monitr.exe
[toed] DXX:<7.EXEC>! i mem
```

503. pages, Entry vector loc IF\%ERR+162 len 5

Section 0	R, W, E, Private			
Section 1	R, W, E, Private			
1000-1123	TOED:<SYSTEM>MONITR.EXE.103052	1-124	R, CW, E	
1131-1135	TOED:<SYSTEM>MONITR.EXE.103052	125-131	R, CW, E	
1433-1634	TOED:<SYSTEM>MONITR.EXE.103052	132-333	R, CW, E	
Section 5	R, W, E, Private			
5001-5252	TOED:<SYSTEM>MONITR.EXE.103052	334-605	R, CW, E	
5755-5777	TOED:<SYSTEM>MONITR.EXE.103052	606-630	R, CW, E	
Section 6	R, W, E, Private			
6433-6465	TOED:<SYSTEM>MONITR.EXE.103052	631-663	R, CW, E	
6506-6572	TOED:<SYSTEM>MONITR.EXE.103052	664-750	R, CW, E	
6577-6615	TOED:<SYSTEM>MONITR.EXE.103052	751-767	R, CW, E	

Figure A.3: Memory utilization of a Tops-20 monitor.

Appendix B

A Debugging Suggestion

Note: this appendix was written before the rest of this document, but a version of the debugger described here was used in the μ Toad. Non- μ Toad-specific sources with configurable widths and depths are available.

Micromachines can make complex designs more simple, but infrequent errors may be hard to catch. One must prepare to catch the error by storing the micromachine state. With a small amount of additional logic, one can build error-catching circuitry into the design, simplifying the debugging process.

B.1 The problem

To debug a micromachine, we need to store its execution flow. We usually do this by connecting a logic analyzer to the jump address bus coming out of the microstore. We set the logic analyzer to trigger on certain conditions—often an error jump location. We make sure that the logic analyzer stores state before the trigger, so that we can see how we got to the trigger location.

However, we must be expecting the error. We must connect the logic analyzer and arm it so we store addresses before the error, and so that the analyzer triggers on the error. If something is not set up correctly, we must restart the process to try to catch the error again. In some cases, waiting for the next instance of the error takes a long time.

Since we are triggering on the error jump location, our micromachine already knows when there is an error. If we had a way to continuously store a small amount of state before the micromachine detects an error, and a way to read this state out of the chip, we might be able to debug some of these errors the first time they occur.

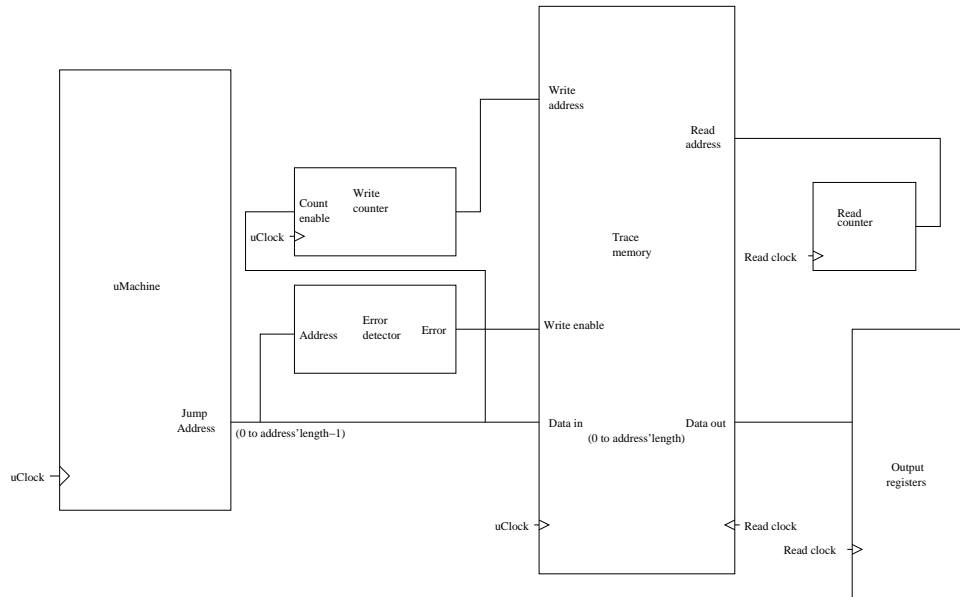


Figure B.1: Trace engine block diagram.

B.2 A solution

We want a lightweight way to store an execution trace and read it out some time after the error occurs. We will call this a *trace engine*.

One way a trace engine might work is as follows: we have a dual-port circular buffer. Every time we execute a microinstruction, we store the address in the next location in the circular buffer. When an error is detected, we mark and store the error address in the buffer, and stop storing any more.

The other side of the circular buffer is continuously reading. When we know we have an error, we connect our logic analyzer to the read side of the circular buffer. We capture the contents of the circular buffer, watching for the address we marked as the last one stored. We can repeat this process as many times as necessary, since the buffer is continuously being read.

A block diagram of the basic system is shown in figure B.2. A description of each significant piece is given below:

Error detector This could be an error bit coming directly from the microstore, or it could be a detector for error jump locations in the microstore. It should delay the error detection one cycle, so that we capture one cycle of the error location.

Write counter This counter controls the write address into the trace memory. It runs continuously, wrapping around, until an error is detected. When an error is detected, it halts.

Element	LUTs	Registers	BRAMs	IOs	Comments
Error detector	4	1	0	0	for a 13-bit detector
Write counter	9	9	0	0	512-address counter
Trace memory	0	0	1	0	512x36
Read counter	9	9	0	0	512-address counter
Output	0	14	0	14	13 address bits, 1 error bit
Total	22	33	1	14	

Figure B.2: Resources required for a 13-bit block-ram-based trace engine.

Trace memory This dual-port memory stores the execution trace on its write port, and reads the trace out for debugging on its read port. For each microcycle, it stores the current address and the error detector value, so that the end of the trace can be identified.

Read counter This counter controls the read address for the trace memory. It runs continuously and wraps around.

Output registers The data read from the trace memory will likely be registered on its way out of the chip.

Potential logic element utilization for a block-RAM-based micromachine with a 13-bit address is shown in figure B.2. These are hypothetical, not actual results.

B.3 Notes

The use of dual-port RAM or SRL16s allows the read clock to be different than the write clock, so that a 66MHz logic analyzer may read a trace from a 133MHz micromachine. If one captures after the write counter has stopped, there are no read/write contention issues.

If the memory is wide enough, it may address itself, eliminating the read counter. For instance, a single block ram in 512x36 mode might use 9 of the 36 data bits as a jump address field of sorts, with each location pointing to the next location (0 to 1, 1 to 2, etc.). We might make this work on the write port as well with a small amount of work.

If a trace depth of 16 was acceptable, the shift register functionality of the LUTs could be used for storing the trace. This would eliminate the write counter from the design. This would require very little logic: one LUT (as a shift register) for each bit of the jump address, one for the error detection bit, as many LUTs and registers as are needed for the error detector, as many LUTs and registers as are needed for the read counter, and as many registers as are needed for the data output. For a micromachine with a seven-bit address and no error bit, we would need to use 15 LUTs and 13 registers, as detailed in figure B.3.

Element	LUTs	Registers	IOs	Comments
Error detector	3	1	0	for a 7-bit detector
Trace memory	8	0	0	7 address bits, 1 error bit
Read counter	4	4	0	16-bit counter
Output registers	0	8	8	7 address bits, 1 error bit
Total	15	13	8	

Figure B.3: Resources required for a 7-bit SRL16-based trace engine.

Extra bits may be stored in the trace memory to aid in debugging, but many things may be deduced from the execution trace.

One might make the read and write counters resettable, so that the logic analyzer could capture the jump address values as they occur until an error is detected. We could capture a longer trace if we had the logic analyzer set up prior to the error—just as we do with our current debugging practices. After an error, the read address would continue to run, looping through the trace captured just before the error, allowing us to capture the data again if necessary.

Chipscope may be used to read data from the trace memory instead of driving the data out of the chip to a logic analyzer. Alternatively, software could be written to read data from the trace memory via User JTAG instead of driving the data out of the chip to a logic analyzer. The software might even parse some microassembler files and show, textually or graphically, the values of fields in the trace memory.

If the trace engine slows down the design, or if it takes up too much space, it can be removed in a production version.

B.4 Conclusion

With a small amount of logic, we can build debugging hardware into our designs that is always watching for errors, simplifying our debugging process.