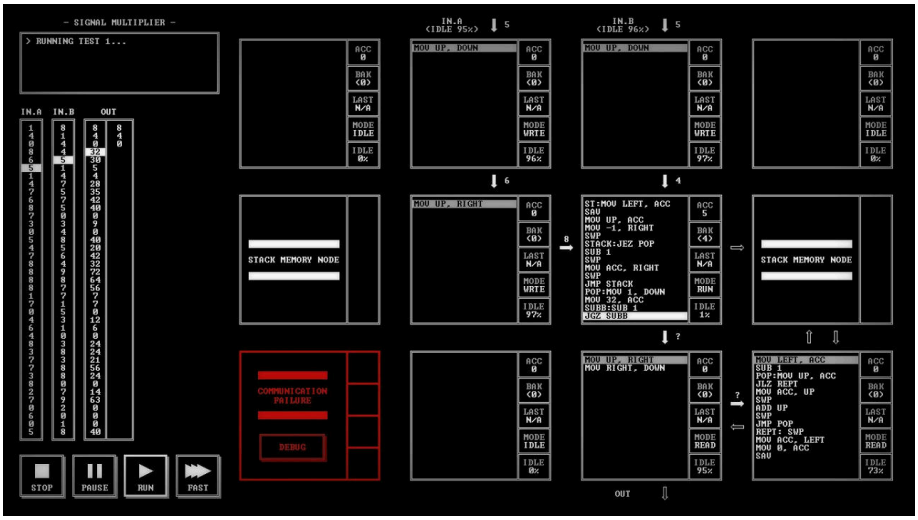
# Running TIS-100 on an FPGA




I successfully implemented TIS-100 in VHDL. The nodes can be programmed via a memory-mapped slave interface to store instructions and perform I/O with the system. Additionally, I wrote a custom assembler in C to encode and decode instructions.

In the first part of the assignment, I managed to instruct the TIS system on the Nios II processor running a MicroC RTOS to execute instructions from JTAG Serial. I also designed a hardcore system with the ARM A9 hardcore, allowing the TIS nodes to be instructed on Linux via a Rust program. In the second part of the assignment, I utilized an existing TIS-100 implementation in C to perform benchmarks and compare hardware and software.

The VHDL implementation can be found on GitHub [1]. The repository contains all the VHDL and C code.

## Contents

# 1. Writing an assembler

In the proposal, I developed the following instruction encoding for the 13 instructions within TIS based on the official manual [2]. This was achieved using 16 bits per instruction. The 8 registers and 5 jump conditions (Table 2) are encoded within 3 bits.

| Instruction | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Comment |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| NOP | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| ADD <SRC> | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | SRC | | | |
| SUB <SRC> | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | SRC | | | |
| ADD #<imm10> | 0 | 0 | 0 | 0 | 0 | 0 | imm10 | | | | | | | | | | imm10 is 0 to 999 |
| SUB #<imm10> | 0 | 0 | 0 | 0 | 0 | 1 | imm10 | | | | | | | | | | imm10 is 0 to 999 |
| MOV #<imm10>, <DST> | 1 | 0 | DST | | | imm11 | | | | | | | | | | | imm11 is −999 to 999 |
| MOV <SRC>, <DST> | 1 | 1 | DST | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | SRC | | | |
| JMP | 0 | 1 | 1 | 1 | 0 | 0 | 0 | COND | | | imm6 | | | | | | imm6 is an adres |
| JRO | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | SRC | | | SRC is an offset |
| SAV | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| SWP | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| NEG | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

Table 1: Instruction encoding

| Type Jump | Encoding |
|---|---|
| Onvoorwaardelijk (JMP) | 000 |
| ACC == 0 (JEZ) | 001 |
| ACC < 0 (JLZ) | 010 |
| ACC > 0 (JGZ) | 100 |
| ACC != 0 (JNZ) | 110 |

Table 2: Jump flag encoding

| Register | Encoding |
|---|---|
| NIL | 000 |
| ACC | 001 |
| UP | 010 |
| DOWN | 011 |
| LEFT | 100 |
| RIGHT | 101 |
| ANY | 110 |
| LAST | 111 |

Table 3: Register encoding

Based on this, I can already assemble instructions myself in binary form to load onto the FPGA and configure a TIS node. However, manually encoding instructions is very time-consuming and error-prone. Therefore, I decided to write my own assembler and disassembler in C before implementing the TIS node itself.

Since disassembly is less complex than parsing assembly, I started with that first. This involves reading the opcode and operands from an instruction and translating them into a string. I created a test for this purpose (Snippet 1).

```
static const uint16_t instructions_bin[] = {
    0x0000, 0x01A5, 0x05A5, 0x0806, 0x0C00, 0x8AE8,
    0xA358, 0xE804, 0xD802, 0x4800, 0x5000, 0x4000,
};

static const char *instructions_str[] = {
    "NOP",          "ADD 421",      "SUB 421",       "ADD ANY",
    "SUB NIL",      "MOV 744, ACC", "MOV 856, LEFT", "MOV LEFT, RIGHT",
    "MOV UP, DOWN", "NEG",          "SWP",           "SAV"};

int tis_decode_test() {
    for (int i = 0; i < 12; i++) {
        char buffer[32] = {0};
        tis_decode(instructions_bin[i], buffer);
        printf("%s\n", buffer);
        if (strcmp(buffer, instructions_str[i])) {
            printf("^ Expected %s\n", instructions_str[i]);
        }
    }
}
```

Snippet 1: Dissasembler testcase

After some work I was able to pass all testcases. The next step was to implement the assembler. For this I also created a series of testcases (Snippet 2).

```
char* assembly =
        "START: NOP\n"
        "ADD 421 # TEST\n"
        "TWO: SUB 421\n"
        "ADD ANY\n"
        "SUB NIL\n"
        "MOV 744, ACC\n"
        "JEZ TWO\n"
        "JRO ACC";

    uint16_t expected[] = {
        0x0000, 0x01A5, 0x05A5, 0x0806,
        0x0C00, 0x8AE8, 0x7042, 0x6001
    };
};
```

Snippet 2: Assembler testcase

My first step to a working assembler was to make a function for encoding integers as signed imm11 values, this wasn't too hard.

```c
int tis_imm11_encode(int integer) {
    if (integer < -999) {
        return 999 | imm11_sign_bit;
    } else if (integer < 0) {
        return (integer*-1) | imm11_sign_bit;
    } else if (integer < 999) {
        return integer;
    }
    return 999;
}
```

Snippet 3: imm11 encoding

Writing the assembler functionality quickly became complex due to several considerations:

- Functionality
  - Parsing needed to be done line-by-line, requiring splitting on `\n` and extracting tokens within each line. Initially, I thought `sscanf` could handle this, but `strtok` proved much more suitable.
  - The `strtok` function is not thread-safe and cannot be used when processing strings in separate threads; however, `strtok_r` is thread-safe [3].
  - Parsing is best done in-place to account for limited on-chip memory.
- TIS-100 Behavior
  - Spaces can be replaced with commas (e.g., `MOV,,,NOP,,,NIL`).
  - Writing something like `ADD 2000` is valid but implicitly adds 999.
  - Jump labels and instructions can be on the same line.
  - A maximum of 18 characters per line is allowed.
- Code Quality (For practice and maintainability)
  - Minimize nesting despite the inherent complexity.
  - Avoid hardcoding (e.g., array lengths, masks, etc.).
  - Keep it as simple as possible.

In my first implementation, I forgot that a jump instruction can refer to a label that is only defined later in the assembly. Initially, I immediately searched for the label in an array during a jump instruction to link it to an instruction address (Snippet 4). Not very practical.

```
JMP END # ERROR: END NOT FOUND
END: NOP
```

Snippet 4: Declaration of label after referencing

Jump instructions should only be linked to labels after the entire program has been parsed. In the new code, I store all labels in two separate arrays, which are later linked together (Snippet 5).

```
char *labels_pos[16] = {0}; // Positions of labels
char *labels_ref[16] = {0}; // References to labels

// Parse instructions...

// Link jump labels
for (int ref_pc = 0; ref_pc < (sizeof(labels_ref) / sizeof(labels_ref[0])); ref_pc+
+) {
    // Skip empty
    if (labels_ref[ref_pc] == NULL) {
        continue;
    }

     for (int pos_pc = 0; pos_pc < (sizeof(labels_pos) / sizeof(labels_pos[0]));
pos_pc++) {
        // Skip empty
        if (labels_pos[pos_pc] == NULL) {
            continue;
        }

        // Check if labels match
        if (strcmp(labels_pos[pos_pc], labels_ref[ref_pc]) == 0) {
            // Edit instruction referencing label
            instructions[ref_pc] |= pos_pc & imm6_mask;
        }
    }
}
```

Snippet 5: Linking of labels

# 2. VHDL Implementation TIS node

## 2.1. Avalon Interface

After completing the assembler, I started developing the TIS node itself. Here too, I chose to first create a testbench to enable quick validation of my design. In this testbench, a series of instructions are written via the memory-mapped slave interface, and I check whether they are correctly stored in the component (see Snippet 6). For this component, I used an example from the course CSC10 as a basis [4]. The full register map is shown in Table 4.

| Address | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Purpose |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0x0 | uint16_t last instruction index | | | | | | | | | | | | | | | | Config |
| 0x2 | uint16_t instruction | | | | | | | | | | | | | | | | Instruction |
| 0x4 | uint16_t instruction | | | | | | | | | | | | | | | | Instruction |
| 0x6 - 0x1F | uint16_t instruction | | | | | | | | | | | | | | | | Instruction |

Table 4: Execution node memory mapped registers

```
node: tis_execution_node
    port map (
      clock          => clock_tb,
      resetn         => resetn_tb,
      read           => read_tb,
      write          => write_tb,
      chipselect     => chipselect_tb,
      address        => address_tb,
      readdata       => readdata_tb,
      writedata      => writedata_tb,
      byteenable     => byteenable_tb,
      Q_export       => Q_export_tb,
);

-- Node Header (15 downto 0)
-- 0 NOP (31 downto 16), (Prefix number is PC)
write_tb <= '1';
address_tb <= std_logic_vector(to_unsigned(0, address_tb'length));
writedata_tb <= x"0000" & x"0006";
ClockPulse(clock_tb);

-- Validate memory state
ClockPulse(clock_tb);
assert readdata_tb = x"0000" & x"0006" report "(0) Failed to validate memory"
severity error;
address_tb <= std_logic_vector(to_unsigned(1, address_tb'length));
```

Snippet 6: Memory mapped slave testbench

## 2.2. Behavior of addressing with ANY

After validating the Avalon interface, I decided to think about how I/O could be implemented within the TIS system, as this fundamentally determines the processor's design.

In TIS-100, the `ANY` register allows addressing all surrounding nodes. However, this does not mean that multiple nodes can receive the same value simultaneously; only one node will receive the value. The mechanism determining how this is evaluated is not documented in the game, nor is there a good explanation available online. I decided to experiment myself to uncover how this mechanism works.

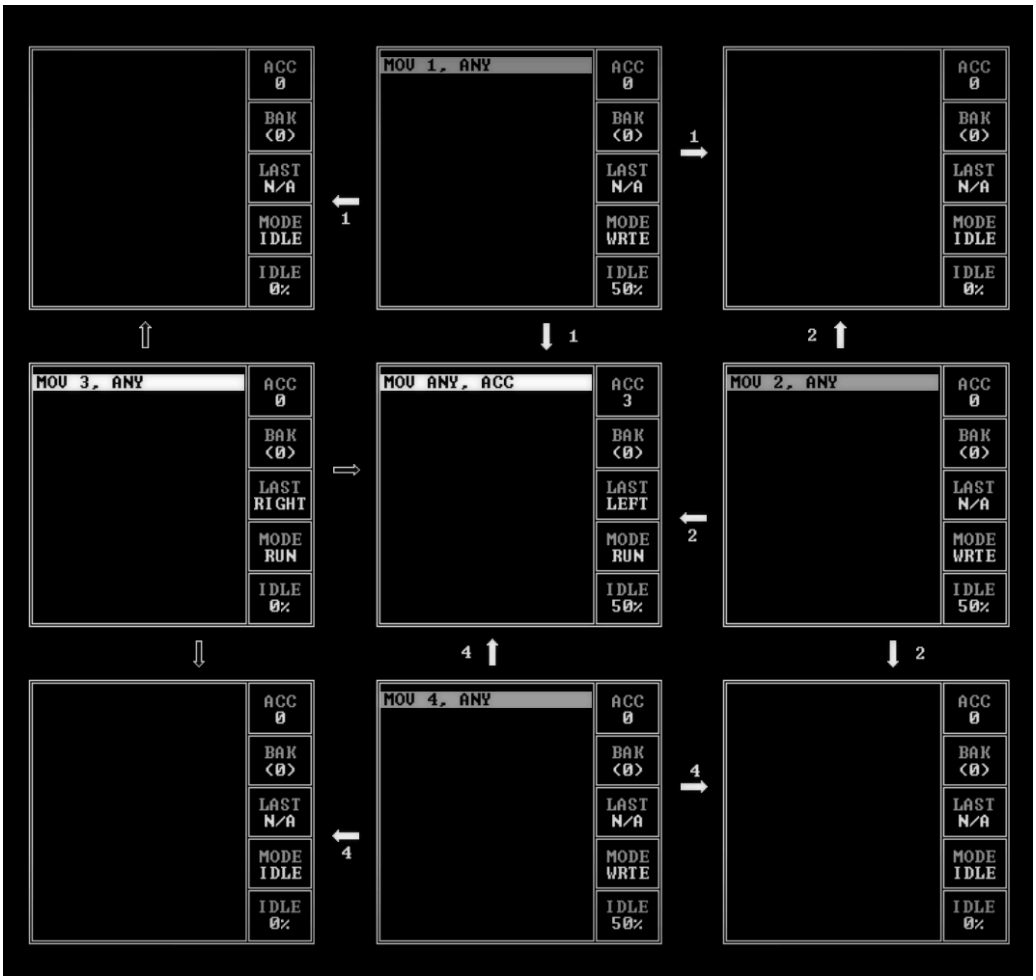



Figure 1: Priorities when reading using ANY in TIS-100

In the example in Figure 1, the node in the middle can read from any of the surrounding nodes. TIS-100 prioritizes `LEFT` first, then `RIGHT`, followed by `UP`, and finally `DOWN`. As a result, the middle node reads `3` from the left node.

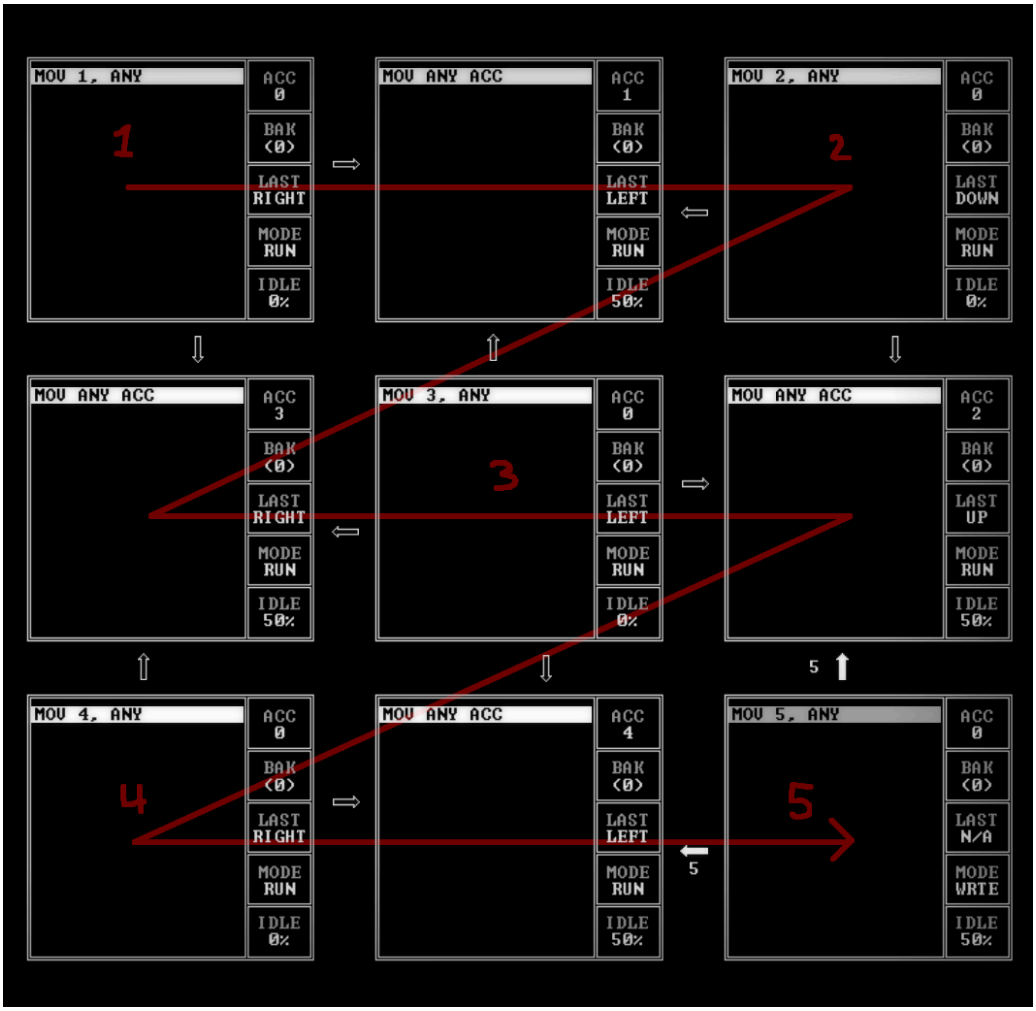



Figure 2: Priorities when writing using ANY in TIS-100

In the example from Figure 2, the write order when addressing with `ANY` becomes apparent. In TIS-100, the top-left node executes first and writes `1` to the right node because `RIGHT` has higher priority than `DOWN`. Next, the top-right node is evaluated, and it writes to `DOWN` since `LEFT` is occupied and no other options are available. Then, the middle node is evaluated, writing `3` to `LEFT` as it has the highest priority. After that, the bottom-left node writes to `RIGHT`. Finally, the bottom-right node is evaluated but cannot write to any surrounding nodes since all are already occupied.

This suggests that the writing nodes determine where the value is delivered and that writing nodes are evaluated from left to right and then from top to bottom.


Figure 3: Evaluation order in TIS-100

In Figure 3, something quite different happens: the left-middle and bottom-right nodes do not receive a value. This indicates that the write priority is not solely determined by the writing nodes, or else the left-middle node would have written to the bottom-left node.

When writing, `UP` is given priority first, then `LEFT`, followed by `RIGHT`, and finally `DOWN`. However, it seems that a writing node may only check `UP` and `LEFT`, and the fact that `RIGHT` and `DOWN` receive priority afterward could be a result of reading nodes being evaluated later, which then request the value (Snippet 7). This is the only explanation I can find to account for all examples.

```
class Node:
    state: enum
    src: int
    dst: int

def read_ANY(node: Node):
    for adjescant in [node.LEFT, node.RIGHT, node.UP, node.DOWN]:
        if adjescant.state == WRITE_ANY:
            adjescant.dst = node.src
            node.state, adjescant.state = (RUNNING, RUNNING)
            return

def evaluatePorts():
    for row in rows:
        for node in row:
            if node.state == READ_ANY:
                read_ANY(node)
```

Snippet 7: Evaluation mechanism TIS-100 in pseudocode

## 2.3. Implementation for addressing with ANY

The good news is that I've been able to clearly understand how TIS-100 evaluates the `ANY` port. Each reading node is evaluated one by one. This happens column by column, and then row by row. The bad news is that this mechanism relies on sequential behavior, meaning that the nodes cannot actually operate in parallel.

Take, for example, the following interface:

```
-- Left conduit
i_left_read   : in  std_logic; -- High when LEFT wants to read from this node
i_left_write  : in  std_logic; -- High when LEFT writes to this node
i_left        : in  in  std_logic_vector(10 downto 0); -- Read data
o_left_read   : out std_logic; -- High when this node wants to read LEFT
o_left_write  : out std_logic; -- High when this node writes to LEFT
o_left        : out in  std_logic_vector(10 downto 0); -- Write data
-- Right conduit
i_right_read  : in  std_logic; -- High when RIGHT wants to read from this node
i_right_write : in  std_logic; -- High when RIGHT writes to this node
i_right       : in  in  std_logic_vector(10 downto 0); -- Read data
o_right_read  : out std_logic; -- High when this node wants to read RIGHT
o_right_write : out std_logic; -- High when this node writes to RIGHT
o_right       : out in  std_logic_vector(10 downto 0); -- Write data
```

Snippet 8: I/O Conduit interface

Suppose a node with these two conduits wants to read a value using `ANY`. This will cause both `i_left_read` and `i_right_read` to be set high. In the next clock cycle, the surrounding nodes will notice this and send a pulse back so the node knows it can read the value. However, the node will only read the left value, and the right value will not be used. At this point, the right node cannot know whether its value has actually been used.

Therefore, a mutual consensus mechanism is needed for transferring data with `ANY`. The question is, how?

### 2.3.1. State consensus

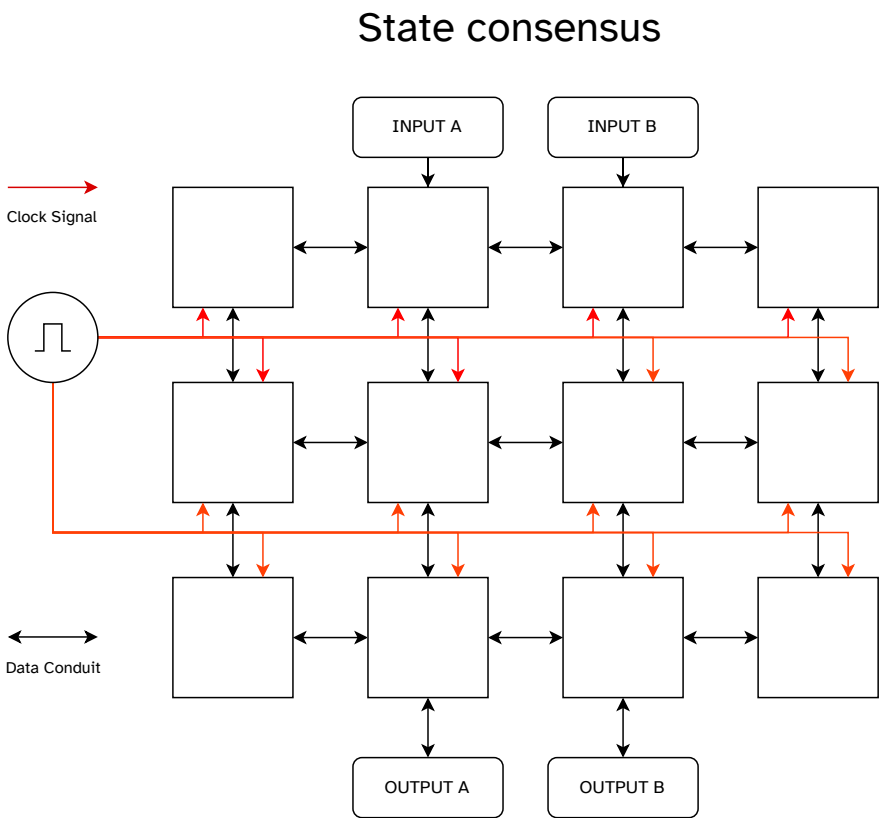



Figure 4: Signals using state consensus

It is possible to use a consensus mechanism based on shared state where all nodes only have visibility of their adjacent nodes. In this implementation, 6 states are used per node evaluation:

1. Load instruction
2. Read `LEFT` with `left_active`, write `RIGHT` with `right_active`
3. Read `RIGHT` with `right_active`, write `LEFT` with `left_active`
4. Read `UP` with `up_active`, write `DOWN` with `down_active`
5. Read `DOWN` with `down_active`, write `UP` with `up_active`
6. Complete instruction based on I/O result

In states 3-6, it is checked whether both `left_active`/`right_active` or `up_active`/`down_active` are high to determine if the I/O operation was successful.

Advantages:

- The connection between nodes can remain simple
- No increase in propagation delay when adding more nodes
- Each node can continue using the same clock signal

Disadvantages:

- The implementation is not 1:1 with the behavior in TIS-100
- Five clock cycles are required per node evaluation

### 2.3.2. Combinational consensus

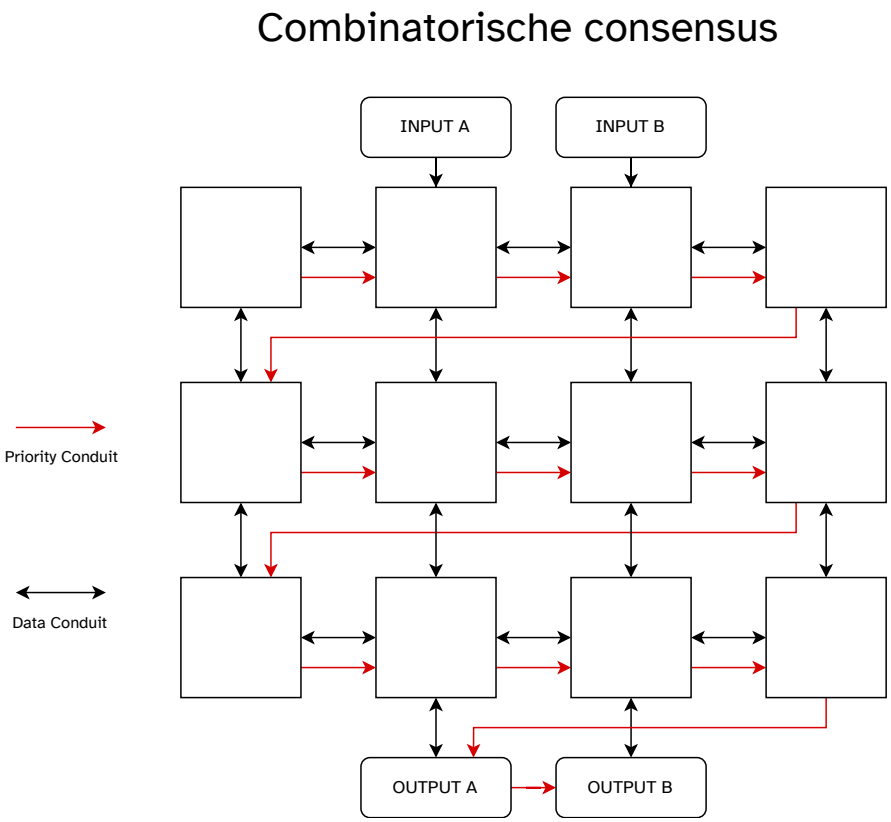



Figure 5: Signals for combinational consensus

It is also possible to use a combinatorial consensus mechanism where the nodes communicate with each other directly, without extra steps. In this approach, the node with the highest priority gets the chance to read a value.

Advantages:

- The behavior within TIS-100 can be replicated 1:1.

Disadvantages:

- It is complex to implement and debug.
- More signals are required to make the system function.

### 2.3.3. Decision

I ultimately decided to use state consensus within the nodes. Although this means the implementation can no longer be 1:1 with the original behavior, I had more confidence that I could successfully realize this design. Additionally, using this method allowed me to maintain the original node conduit interface, simplifying the implementation. This resulted in the state machine shown in Figure 6.

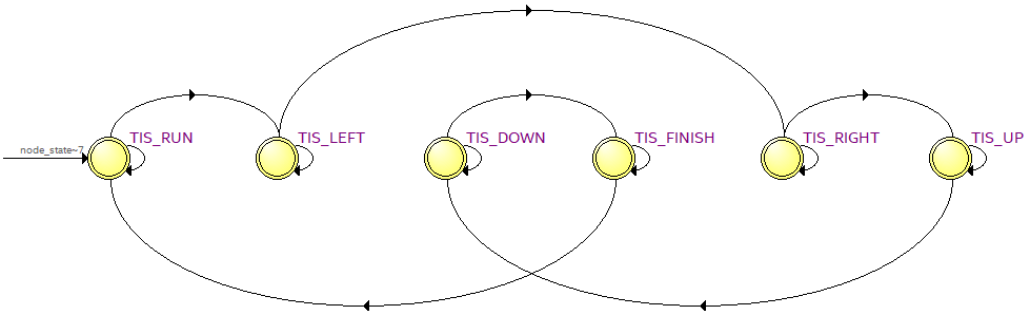



Figure 6: TIS State machine

## 2.4. Implementing instruction execution in TIS nodes using VHDL

Earlier, I created a testbench to test the memory-mapped slave interface, which stores instructions within the TIS node. The next step was to expand the testbench to also validate the execution of instructions. For this, I wrote a short program covering multiple instructions (Snippet 9). In the test, I check the state of the ACC and the program counter after each instruction.

```
NOP
ADD 421
SUB 421
ADD ANY
SUB NIL
MOV 744, ACC
JNZ 8
MOV 123, ACC
MOV 456, ACC
```

Snippet 9: Instructions used for testing

For each instruction, 6 rising edges are generated within the testbench. During the first rising edge, the instruction is read, the next 4 clock cycles are used for I/O with the surrounding nodes, and in the final cycle, the internal state (ACC, PC, BAK) is updated.

Eventually, I was able to implement all the instructions, but my implementation is not very clean. I opted for a brute-force approach using if-statements instead of a separate component for instruction decoding and the ALU. There is a lot of nesting within the if-statements, so it doesn't fit within this document. [1]

To gain more control over the instruction output, I also created a component that captures the instruction output at the end of each instruction. This component can also be connected to a button/signal to execute instructions step by step. In Platform Designer, a reset interface is used to pass the signal.

## 2.5. Implementing memory mapped I/O with the TIS system using VHDL

In Section 2.4, I succeeded in setting up a functional TIS-100 node. However, external I/O for transferring values with the system still needs to be realized.

Directly using a memory-mapped slave to write to a node's port is not ideal, as values must always be written to the system in the correct order and without delay. Therefore, I achieve this through a separate 'stack memory node,' which can also be found in the game (Figure 7). This node can be connected to execution nodes, just like in Snippet 8.

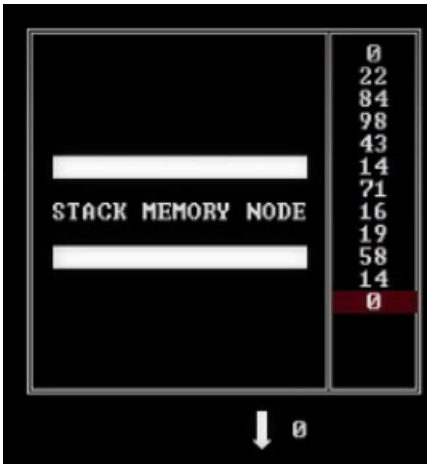



Figure 7: TIS-100 Stack Memory Node

The advantage of this node is that it has a buffer that can write values to the system without delay and in the correct order. The surrounding nodes continuously read/write values from the bottom of the stack, while values are read/written from the top via the memory-mapped slave. This ensures that values are read in the correct order within the system. Internally, it functions more like a circular buffer ([5]) rather than a stack.

The memory map of the stack node can be seen in Table 5:

| Address | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Purpose |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0x0 | Element Count | | | | | | | | Unused | | | | | | I | O | Configuration |
| 0x2 | int16_t | | | | | | | | | | | | | | | | Reading/writing |

Table 5: Stack node memory mapped registers

- 8 bits for the current number of values in the buffer
- Bit 'I' determines if the stack is reading from surrounding nodes
- Bit 'O' determines if the stack is writing to surrounding nodes
- If the buffer is empty, 0xFFFF is read from the node
- When writing a value, it is restricted to the range of −999 to 999

In addition to the memory map, the stack node can also generate an interrupt if there are values in the buffer.

## 2.6. Testing a configuration in Platform Designer using Intel FPGA Monitor

Now that all the important components are set up, I can replicate a scenario from the game. Here, I will use a simple setup where values are directly written from UP to DOWN, as shown in Figure 8.

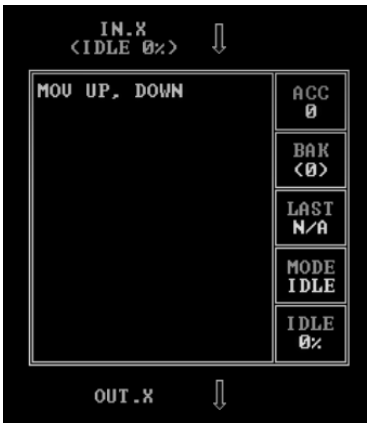



Figure 8: Configuration in TIS-100

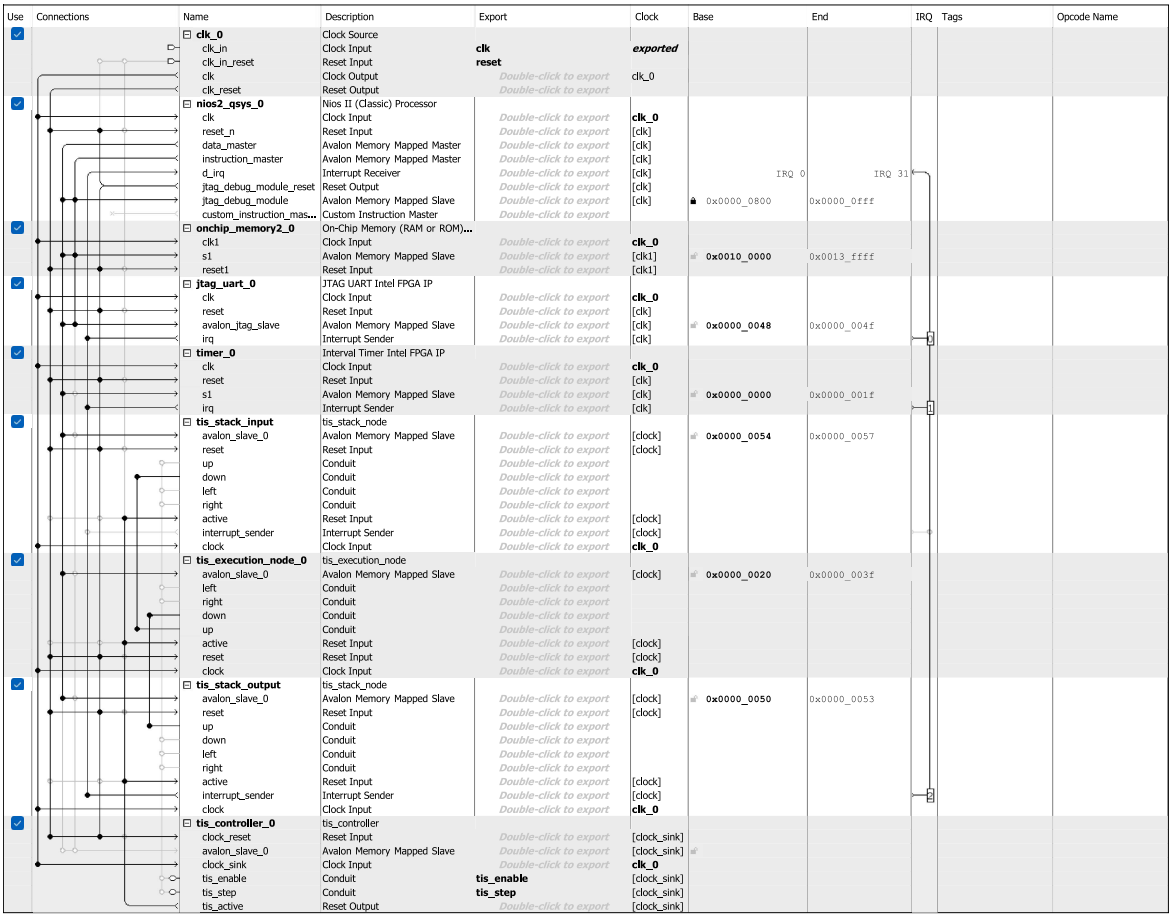



Figure 9: Configuration in Platform Designer

Implementing this scenario in Platform Designer is fairly simple, as the different nodes can be visually connected. UP connects to DOWN, and LEFT connects to RIGHT. If you make an error, you will get an error because the signals have a specific type.

In Figure 9, I use a stack node acting as input, an execution node, and a stack node for output. For the output node, I also connect an IRQ signal to the Nios II processor, which generates an interrupt when a value can be read from the buffer.

To test this setup, I first used the Intel FPGA monitor with a small test program in C (Snippet 10). This makes it very easy to read/write memory and run tests.

```c
int main(void) {
    *node_config = 0; // 0 is last instruction index
    node_instruction[0] = 0xD802; // MOV UP, DOWN

    // Provide inputs
    *input_stack_value = 1;
    *input_stack_value = 2;
    *input_stack_value = 3;
    *input_stack_value = 4;

    for (volatile int i = 0; i < 100; i++) {} // Wait

    node_instruction[4] = *output_stack_value;  // Read 1
    node_instruction[5] = *output_stack_value;  // Read 2
    node_instruction[6] = *output_stack_value;  // Read 3
    node_instruction[6] = *output_stack_value;  // Read 4
    node_instruction[8] = *output_stack_value;  // 0xFFFF (Empty)
}
```

Snippet 10: C code to test the system

I needed many attempts to get the setup working due to the complexity of the I/O. I encountered the following issues:

1. I used a ‘read wait’ of 1 in the Avalon slave parameters, which caused the read to stay active for two clock cycles, leading to two reads.
2. I used an incorrect pointer within the circular buffer, causing me to read invalid values
3. I had swapped the order of the steps UP/DOWN in the stack node and the execution node. I only realized this when I opened the state machine viewer.

Eventually, I was able to get the setup working and saw the following output in the FPGA monitor:

```
0x00000010    ?    ?    ?    ?    ?    ?    ?    ?
0x00000020  0000 D802  0000 0000  0000 0001  0002 0003
0x00000030  0004 FFFF    ?    ?    ?    ?    ?    ?
```

Figure 10: **Sucess!**

## 2.7. Implementating a MicroC RTOS system with an assembler

Once I was confident that the I/O with the nodes was functioning correctly, I started writing an RTOS system using MicroC. To avoid delays, I decided to use the same setup as in Figure 9. The difference is that the system can now be programmed and instructed via JTAG serial. The Nios II also converts assembly into machine code for the nodes using the assembler from Section 1, making it possible to write/read values with the stack nodes. The output of this system can be seen in Figure 11, where on the left, an input number is multiplied by 4, and on the right, the number 4 is filtered.

```
Enter an assembly program

MOV UP, ACC

ADD ACC

ADD ACC

MOV ACC, DOWN


Writing program to node
> 0
< 0
> 1
< 4
> 2
< 8
> 3
< 12
> 4
< 16
> 5
< 20
```

```
Enter an assembly program

MOV UP ACC # LOAD

SUB 4 # COMPARE

JEZ EQ

JRO 2

EQ: MOV 4, DOWN

NOP


Writing program to node
> 0
> 1
> 2
> 3
> 4
< 4
> 5
> 6
```

Figure 11: MicroC RTOS demo

Within MicroC, I use two threads:

- One thread to read input from the user
- One thread to display the output from the stack node

The benefit of these threads is that outputs can be shown immediately, and no blocking occurs when entering numbers.

## 2.8. Implementation of hardcore systeem on Linux with Rust

Once I had the TIS-100 system working on an RTOS, the next step was to integrate it into an HPS system with Linux. For this, I made a copy of the setup in Figure 9, thinking that I could simply replace the Nios II with an HPS component. However, this approach failed, as shown in Figure 12. Initially, I thought the system might be too complex to load onto the FPGA with the HPS, but I did not observe any extreme data in the resource usage. I also received an error message after removing all the TIS components in Platform Designer.

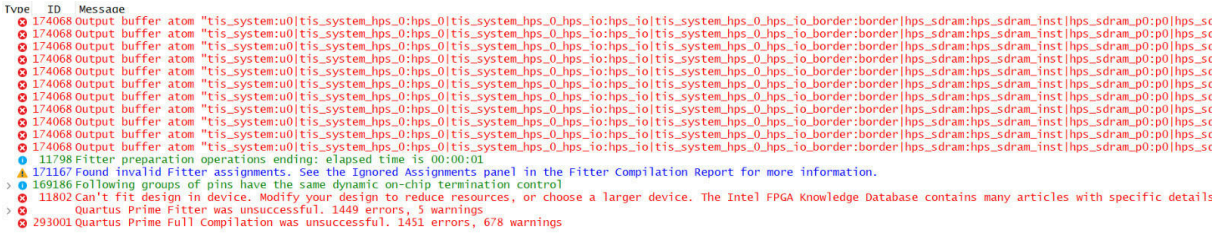



Figure 12: Quartus fitting error using HPS

To solve the problem, I had to use an older project as a basis and added the components there. After that, I did not encounter any issues and was able to successfully set up an HPS system. The exact cause of the issue is still unclear to me.

Next, I needed a way to control the components on Linux. Originally, I planned to do this with a kernel module in C. However, out of curiosity, I tried an implementation in Rust. Linux officially supports Rust now, so I was interested in the development experience.

I had to search a bit for the right steps to get everything working [6], [7]. First, you need to install the correct version of Rust and then compile Rust-for-Linux. Ultimately, I followed the roughly the following steps:

```
[student@csc10 ~]$ curl --proto '=https' --tlsv1.2 https://sh.rustup.rs -sSf | sh
[student@csc10 ~]$ cargo --version
cargo 1.84.0 (66221abde 2024-11-19)
[student@csc10 ~]$ pacman -Syu
[student@csc10 ~]$ pacman -S llvm lld clang unzip
[student@csc10 ~]$ wget https://github.com/Rust-for-Linux/linux/archive/refs/heads/
rust-next.zip # No git clone to save time and disk space
[student@csc10 ~]$ unzip rust-next.zip
[student@csc10 ~]$ cd linux-rust-next/
[student@csc10 linux-rust-next]$ rustup override set $(scripts/min-tool-version.sh
rustc)
[student@csc10 linux-rust-next]$ sudo pacman -S rust-src rust-bindgen
[student@csc10 linux-rust-next]$ rustup component add rust-src # Rust standard
library source
[student@csc10 linux-rust-next]$ make LLVM=1 rustavailable
Rust is available!
[student@csc10 linux-rust-next]$ cp /usr/lib/modules/$(uname -r)/build/.config .
[student@csc10 linux-rust-next]$ make LLVM=1 -j$(nproc) # 4h coffee break!
```

Snippet 11: Setup for Rust kernel development

Initially, I was unsure if I would have enough storage for the development environment, but it turned out to be manageable. After installing everything, there were still 43GB available (Snippet 12). Setting up the environment did take a long time, though. The compilation of Linux took several hours, which is not surprising given the 2 ARM-A9 cores and 1GB of DDR3.

```
Filesystem      Size  Used Avail Use% Mounted on
/dev/root        56G   12G   42G  22% /
```

Snippet 12: Output from `df -h`

Despite successfully compiling the kernel and running `make LLVM=1 rustavailable`, Rust was strangely not activated within the kernel compilation. In `menuconfig`, the Rust samples were nowhere to be found (Figure 13).

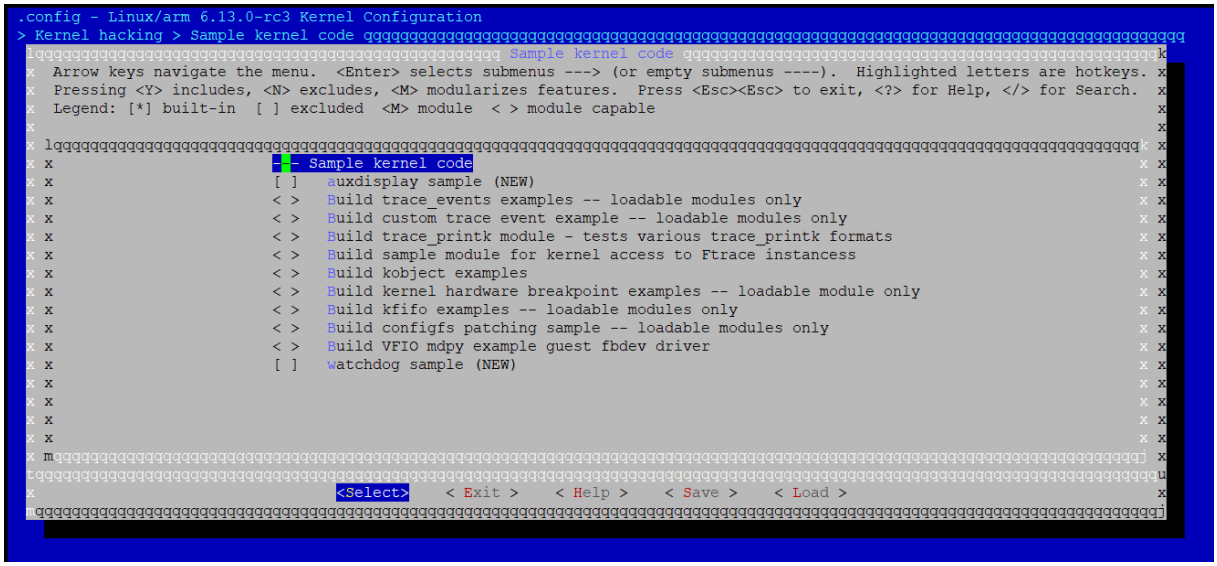



Figure 13: Missing Rust code samples in menuconfig

To resolve the issue, I had to dig through `menuconfig` and identified the following problems:

- `MODVERSIONS` was enabled [=y] and had to be disabled
- `HAVE_RUST` was set to [=n] and had to be enabled

After making these adjustments, Rust support became visible as an option.

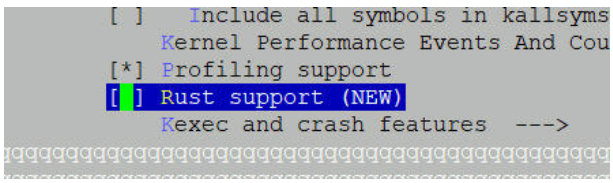



Figure 14: Rust support!

```
ld.lld: error: undefined symbol: __aeabi_uldivmod
>>> referenced by core.442bded086a4b6f8-cgu.0
>>>               rust/core.o:(core::fmt::num::parse_u64_into::<39>) in archive vmlinux.a
>>> referenced by core.442bded086a4b6f8-cgu.0
>>>               rust/core.o:(core::fmt::num::parse_u64_into::<39>) in archive vmlinux.a
>>> referenced by core.442bded086a4b6f8-cgu.0
>>>               rust/core.o:(core::fmt::num::parse_u64_into::<39>) in archive vmlinux.a
>>> referenced 28 more times
>>> did you mean: __aeabi_uidivmod
>>> defined in: vmlinux.a(arch/arm/lib/lib1funcs.o)
make[2]: *** [scripts/Makefile.vmlinux:77: vmlinux] Error 1
make[1]: *** [/home/student/linux-rust-next/Makefile:1226: vmlinux] Error 2
make: *** [Makefile:251: __sub-make] Error 2
[student@csc10 linux-rust-next]$ ld.lld: error: undefined symbol:   aeabi_uldivmod
```

Figure 15: Or not?

Unfortunately, the compilation still fails. After further investigation, I concluded that Rust is not supported in the kernel for ARMv7; only the architectures listed in Table 6 are supported. Too bad!

| Architecture | Level of support | Constraints |
|---|---|---|
| arm64 | Maintained | Little Endian only. |
| loongarch | Maintained | - |
| riscv | Maintained | riscv64 only. |
| um | Maintained | - |
| x86 | Maintained | x86_64 only. |

Table 6: Supported Rust Arch [8]

As a plan B, I decided to use a simpler Rust program (Figure 16). In this, nodes are programmed in the same way as in Snippet 10.

```
52      unsafe {
53          ptr::write_volatile(hwreg(NODE_BASE + 0x2), 0xD802); // Write MOV UP, DOWN command
54          ptr::write_volatile(hwreg(INPUT_BASE), 1);
55          ptr::write_volatile(hwreg(INPUT_BASE), 2);
56          ptr::write_volatile(hwreg(INPUT_BASE), 3);
57          ptr::write_volatile(hwreg(INPUT_BASE), 4);
58      }
59
60      // Wait
61      sleep(Duration::from_millis(1));
62
63      for expected in [1, 2, 3, 4, 0xFFFF] {
64          let output: u16 = unsafe { ptr::read_volatile(hwreg(OUTPUT_BASE)) };
65          println!("Got {:X}", output);
66          if output != expected {
67              eprintln!("Error: expected {:X}, got {:X}", expected, output);
68          }
69      }
70
PROBLEMS  OUTPUT  DEBUG CONSOLE  PORTS  TERMINAL                bash - debug
[student@csc10 debug]$ sudo ./tis_rust
[sudo] password for student:
Got 1
Got 2
Got 3
Got 4
Got FFFF
[student@csc10 debug]$
```

Figure 16: Configuring nodes in Rust

# 3. Comparing a C and VHDL implementation

## 3.1. Implementation TIS-100 systeem in C

Due to time constraints, I wasn't able to write my own C implementation of the TIS system. As an alternative, I chose to use an existing C emulator [9] to still make a comparison between C and VHDL.

## 3.2. Comparing C en VHDL implementation

In the comparison, I will determine the difference in execution speed between emulating TIS-100 on the hard core and running programs on the FPGA. The expectation is that emulation in simple cases (e.g., `MOV UP, DOWN`) won't be drastically faster due to the higher clock speed, but with a higher number of nodes and more mathematical operations, it will be significantly slower. The CPU must manually perform bounds checking to stay within the range of −999 to 999 and execute everything sequentially, while on the FPGA, everything is done in parallel.

Since my FPGA implementation takes the same number of cycles as the game to execute a program, I will work out and execute programs from the game in my comparison to determine the number of cycles.

In the game, TIS-100 can operate at a maximum of 5KHz. My FPGA implementation completes a full node cycle in 6 clock cycles. The FPGA runs at 50MHz, so the TIS clock speed is 8.33MHz. Execution times will be calculated as shown in Figure 17.

$$t = \frac{\text{cycles}}{8.333 * 10^6}$$

Figure 17: Calculating execution time

For the output in C, I will perform a timing measurement to determine how long the execution took. The implementation of the measurement is shown in Snippet 13. During the execution of the program, no printing takes place to avoid additional overhead from I/O.

```c
printf("Starting measure\n");
  int cycles = 0;
  clock_t begin = clock();
  int all_blocked_count = 0;
  while (all_blocked_count < 5) {
    int all_blocked = program_tick(p);
    cycles++;
    if (all_blocked) {
      all_blocked_count++;
    } else {
      all_blocked_count = 0;
    }
  }
  clock_t end = clock();
  double time_spent = (double)(end - begin) / (CLOCKS_PER_SEC/1000);
  printf("Execution took %fms\n", time_spent);
  printf("Execution took %d\n", cycles);
```

Snippet 13: Benchmark in C

## 3.3. Results and conclusion benchmark

| Title | Cycles | Nodes | Instructions | Hardcore | FPGA | Verschil |
|---|---|---|---|---|---|---|
| MOV chain | 83 | 8 | 8 | 0.341 ms | 0.010 ms | 34x |
| Slow MOV chain | 233926 | 8 | 72 | 870.951 ms | 28.071 ms | 31x |
| Division | 6911 | 7 | 37 | 17.501 ms | 0.829 ms | 21x |
| Comparator | 265 | 6 | 20 | 0.703 ms | 0.032 ms | 22x |
| Min/Max detector | 352 | 5 | 33 | 1.374 ms | 0.042 ms | 33x |

Table 7: Benchmark results

Despite the higher clock speed of the ARM A9 (925MHz) compared to the FPGA (50MHz), it can be seen in Table 7 that the FPGA implementation is at least 20x faster than the hardcore in all benchmarks. What stood out to me is that the difference varies quite a bit between benchmarks. This is likely because the C implementation can skip evaluations of nodes when it is waiting for an input.

Based on these results and the experience I've had programming with VHDL, I think emulation on a hardcore CPU is sufficient in many cases. A difference of 20-30x is not huge, as you could also use a more powerful CPU and the software could be further optimized. Due to caching and pipelining, a CPU can be extremely efficient.

Additionally, energy consumption is lower with a hardcore CPU than with an FPGA. If I needed a TIS-100 implementation that has predictable throughput like the FPGA, it is also possible to achieve this on a GPU. In terms of energy, I think a GPU would still be more efficient than an FPGA.

Moreover, it is much easier to implement in C/C++ than in VHDL. To create an implementation in VHDL, you need to be much more cautious and test thoroughly to ensure the design is correct.

Only when you need a highly specialized hardware connection where many things happen in parallel does an FPGA implementation seem more sensible. In that case, an FPGA cannot easily be replaced by another piece of hardware or software.

# 4. Conclusion

I was able to fully implement TIS-100 in hardware and integrate it within both an RTOS and a Linux environment. Initially, it was not my plan to implement the stack node from the game (Figure 7), but it turned out to be a good way to realize memory-mapped I/O. The only thing I didn't manage to do was create my own TIS-100 implementation in C, but with more time, I'm sure I could have done that as well.

The most challenging part was implementing the I/O between nodes; I ran into a lot of edge cases. This ended up taking me more time than the assembler and instruction encoding combined. I now feel I have a much better understanding of how signals function in hardware.

Setting up tests was also extremely helpful in performing large refactors within the code without accidentally introducing errors. You have to adopt a completely different mindset with VHDL than with software to create a well-functioning design.

# Bibliography

[1] "TIS-100-FPGA." [Online]. Available: https://github.com/Powerbyte7/TIS-100-FPGA

[2] Zachtronics, "TIS-100 Reference Manual." [Online]. Available: https://www.zachtronics.com/images/TIS-100P%20Reference%20Manual.pdf

[3] "strtok() and strtok_r() functions in C with examples." [Online]. Available: https://www.geeksforgeeks.org/strtok-strtok_r-functions-c-examples/

[4] "32-bit Avalon Interface CSC10." [Online]. Available: https://bitbucket.org/HR_ELEKTRO/csc10/src/master/progs/reg32_avalon_interface.vhd

[5] Wikipedia contributors, "Circular buffer — Wikipedia, The Free Encyclopedia." [Online]. Available: https://en.wikipedia.org/w/index.php?title=Circular_buffer&oldid=1263915242

[6] "Rust Kernel Video: Getting Started." [Online]. Available: https://wusyong.github.io/posts/rust-kernel-module-00/

[7] "Mentorship Session: Writing Linux Kernel Modules in Rust." [Online]. Available: https://youtu.be/-l-8WrGHEGI

[8] "Linux Kernel Docs - Rust Arch Support." [Online]. Available: https://docs.kernel.org/6.13/rust/arch-support.html

[9] R. Ward, "TIS-100 C Emulator." [Online]. Available: https://github.com/eviltrout/tis-100/

# 5. Bijlage A: Benchmarks


Figure 18: MOV chain (83 cycles, 8 nodes, 8 instructions)

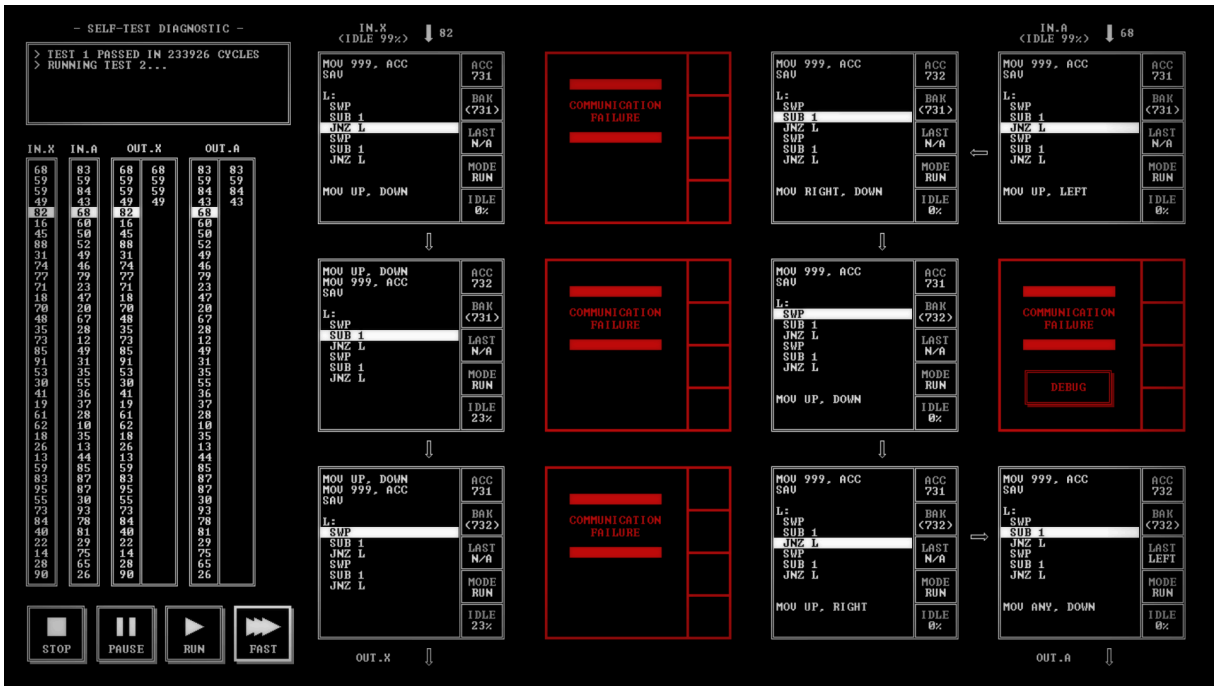



Figure 19: Slow MOV chain (265 cycles, 6 nodes, 20 instructions)

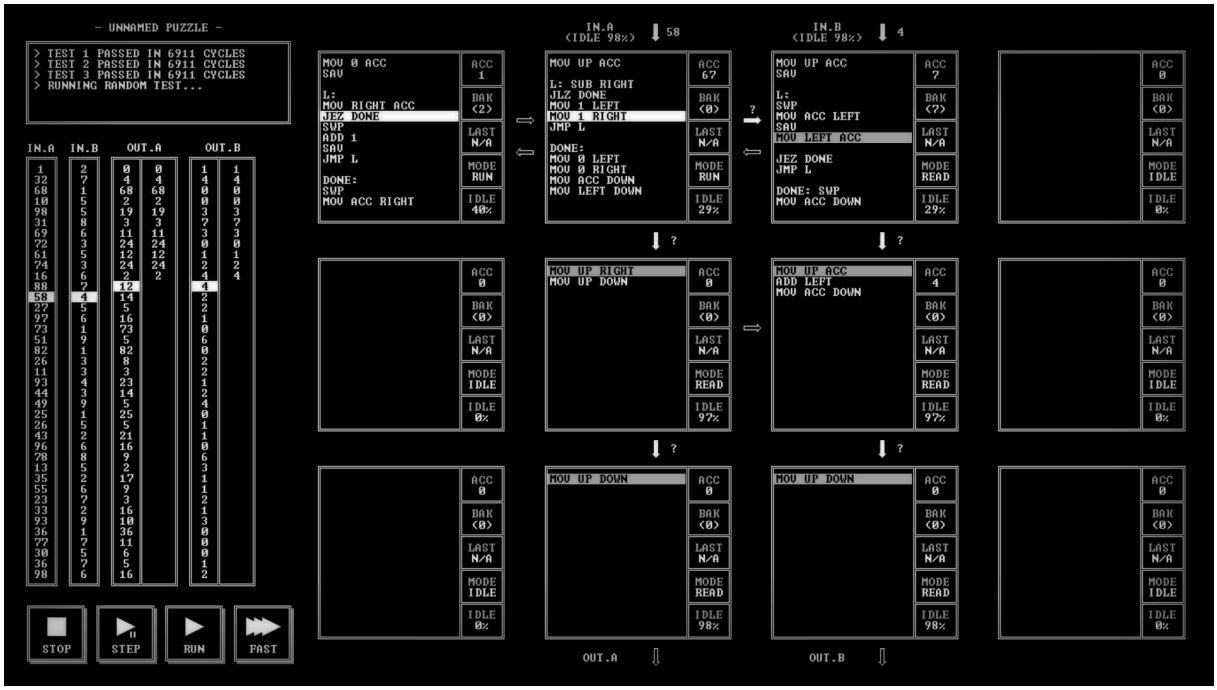



Figure 20: Division (6911 cycles, 7 nodes, 37 instructions)

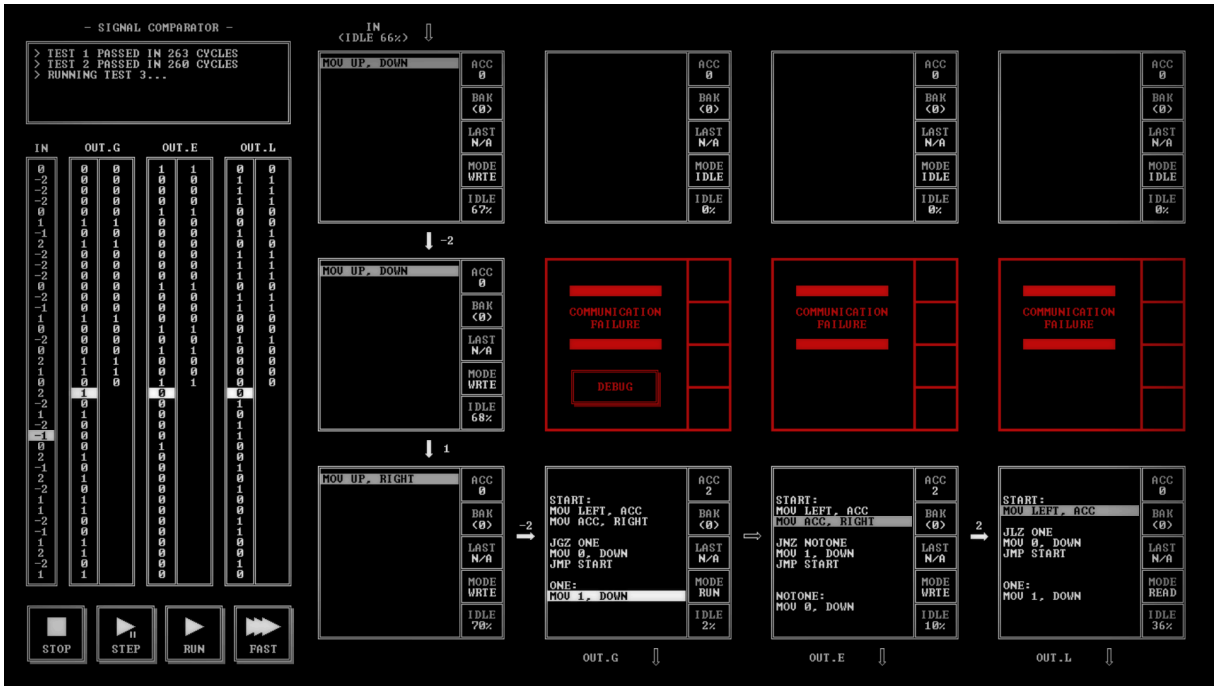



Figure 21: Comparator (265 cycles, 6 nodes, 20 instructions)

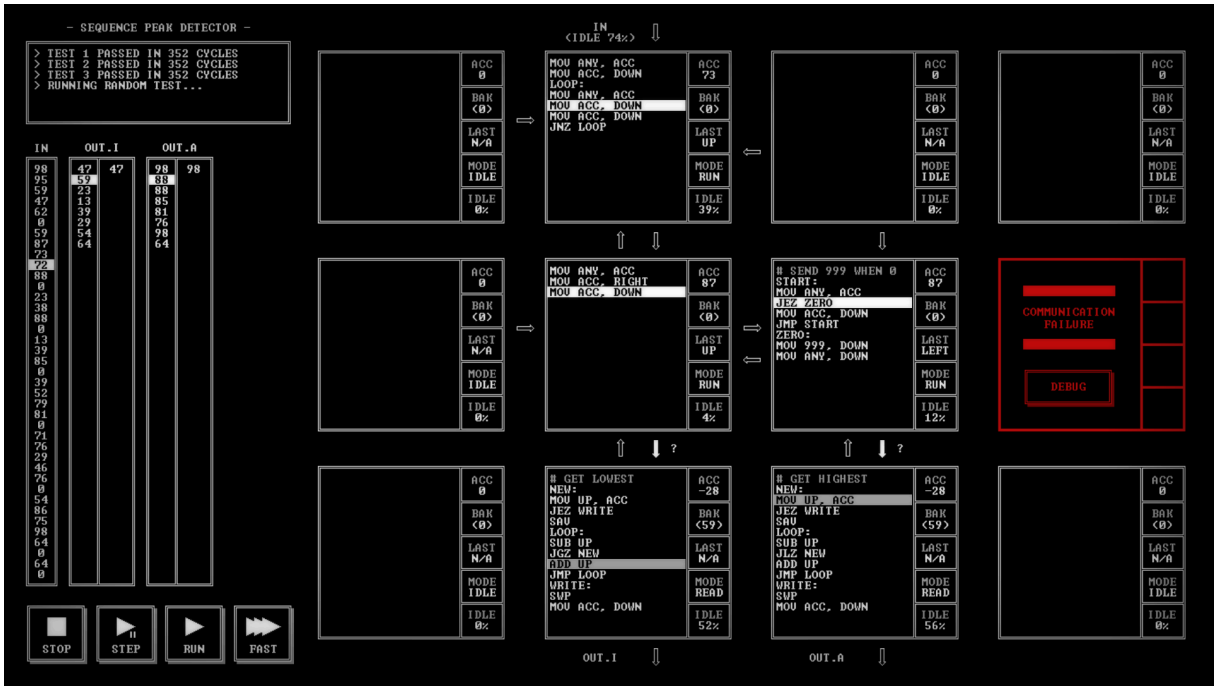



Figure 22: Min/Max detector (352 cycles, 5 nodes, 33 instructions)