DEPARTMENT OF COMPUTER SCIENCE

UNIVERSITY OF OXFORD

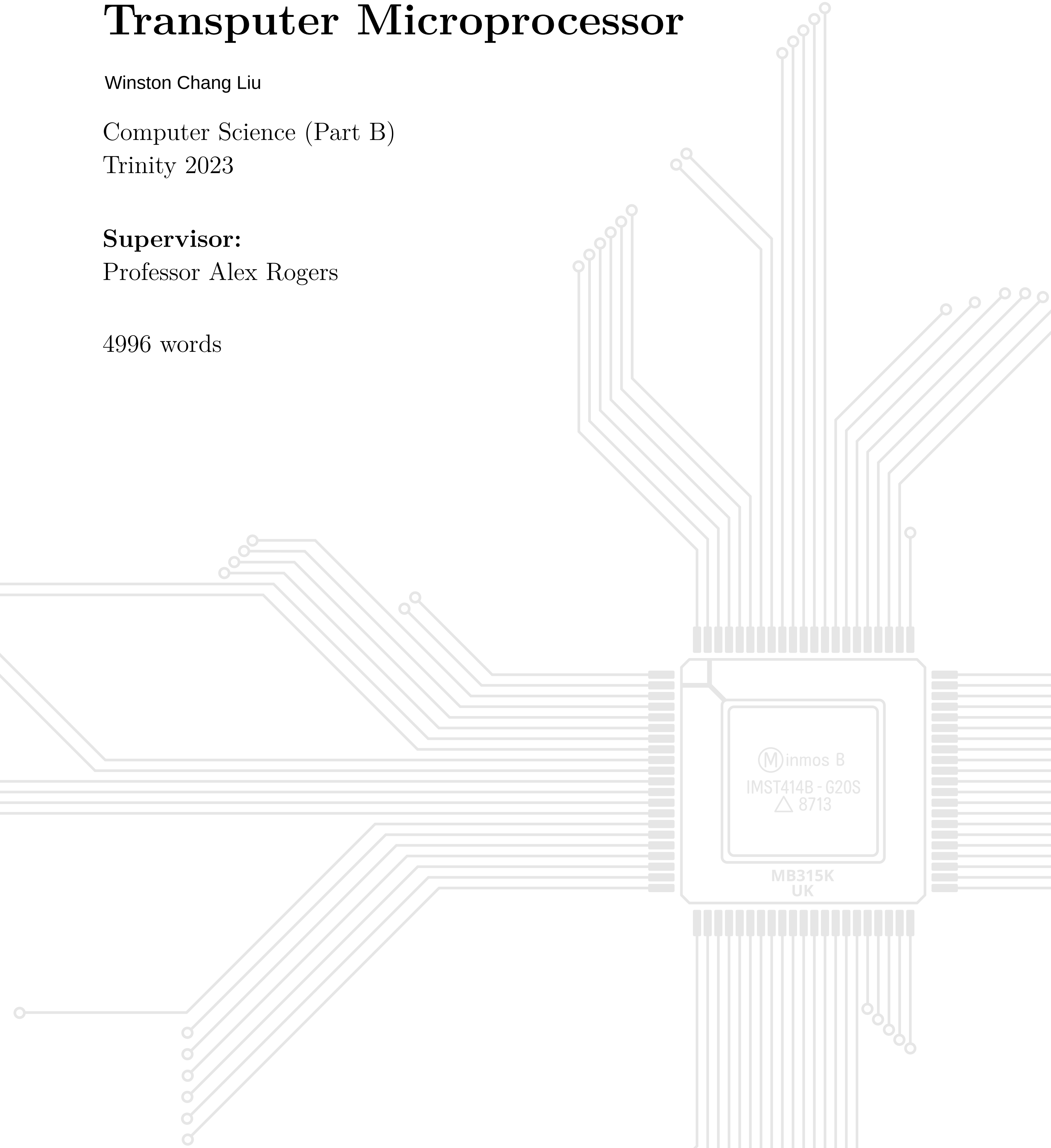
Analysis and Simulation of the Transputer Microprocessor

Winston Chang Liu

Computer Science (Part B)
Trinity 2023

Supervisor:
Professor Alex Rogers

4996 words



Abstract

The Transputer was a class of processors released by INMOS during the 1980s that provided hardware support for concurrent processes. This project investigates the features of the T414, the first 32-bit Transputer, including its internal stack machine architecture, implementation of internal and external communication channels, support for static and dynamic procedure calls, and conditional flow using guarded alternatives. It also presents a newly standardised Transputer Assembly syntax, alongside the Transputer Assembler and Multi-processor Simulator (TAMS), with which different assembly programming techniques and concurrent tasks are evaluated. By analysing these programs using TAMS, it becomes clear that while there is much value in the concurrency support provided by the T414, its stack machine design severely limits the usefulness of its registers, vastly increasing the frequency of slow memory accesses.

Contents

1 Introduction

The Transputer was a series of microprocessors produced by INMOS under the paradigm of hardware-facilitated concurrency. A core belief of its creators were that through the support of parallel processing within the processors themselves, facilitated by the concurrency-focused *Occam* programming language, the Transputer family of products would adapt to new computing demands for decades to come [1].

These microprocessors, however, have failed to withstand the test of time, losing to the dominance of modern architectures such as x86 and ARM. Nevertheless, their legacy remains, with renewed interest in concurrent systems and parallel computing with the emergence of languages inspired by Occam such as Go[2].

1.1 Contributions

This project takes a deep dive into the inner workings of the T414 Processor, providing concise explanations for its features by extracting its design principals and implementation details from the official documentation.

The project also presents a standardised format for Transputer assembly based on the original instruction set of the T414, along with a new assembler and simulator for the T414, named the Transputer Assembler and Multi-processor Simulator (TAMS for short). TAMS provides explicit support for multi-processor simulations, as well as direct assembly programming and, features not commonly seen in existing simulators [3] [4] [5] [6].

Using Transputer assembly and TAMS, this project demonstrates the ability for networks of interconnected Transputer processors to communicate with each other and complete complex tasks. It also reveals the limitations of its stack registers, which only serve to increase the number of memory accesses, thereby acting as a bottleneck to complex programs.

2 Background

This chapter describes the architecture and instruction set of the first ever 32-bit Transputer microprocessor released by INMOS: the T414. This is the variant on which the assembler and simulator presented in this report is based on.

Many of the details about the Transputer architecture and instruction set have been derived from a publication titled "Transputer Instruction Set: A Compilter Writer's Guide" (hereby refered to as ACWG) [7]. The initial release of the processor family had come with much fanfare surrounding *Occam*, but pressure from the industry eventually gave way to compilers for popular languages such as C and FORTRAN, along with the publication of ACWG [8].

2.1 Sequential Processes

2.1.1 Registers and the Stack Machine

The T414 carried 6 basic registers, 3 of which were general purpose registers used for arithmetic calculations. All registers hold 32-bit values.

Register	Purpose
`Areg`	Evaluation Stack (Top)
`Breg`	Evaluation Stack
`Creg`	Evaluation Stack (Bottom)
`Iptr`	Instruction Pointer (Program Counter)
`Wptr`	Workspace Pointer (Stack Pointer equivalent)
`Oreg`	Operand Accumulator

Table 1: Main Transputer Registers

The 3 general purpose registers organised as part of an evaluation stack, with `Areg` at the top and `Creg` at the bottom.

The stack is treated purely as an evaluation stage, and values deemed useful beyond just single calculations are generally stored in memory. Instructions operate on the stack, like so (Fig. 1):

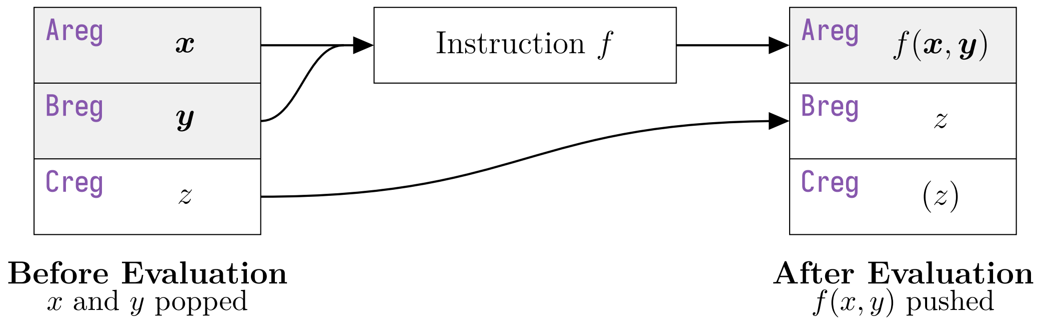


Figure 1: Stack operation involving 2 operands and 1 output value

2.1.2 Memory and Workspaces

The T414 supports up to 4GB of memory, byte-indexed using 32-bit addresses. *Words* are 4 bytes (32 bits) each.

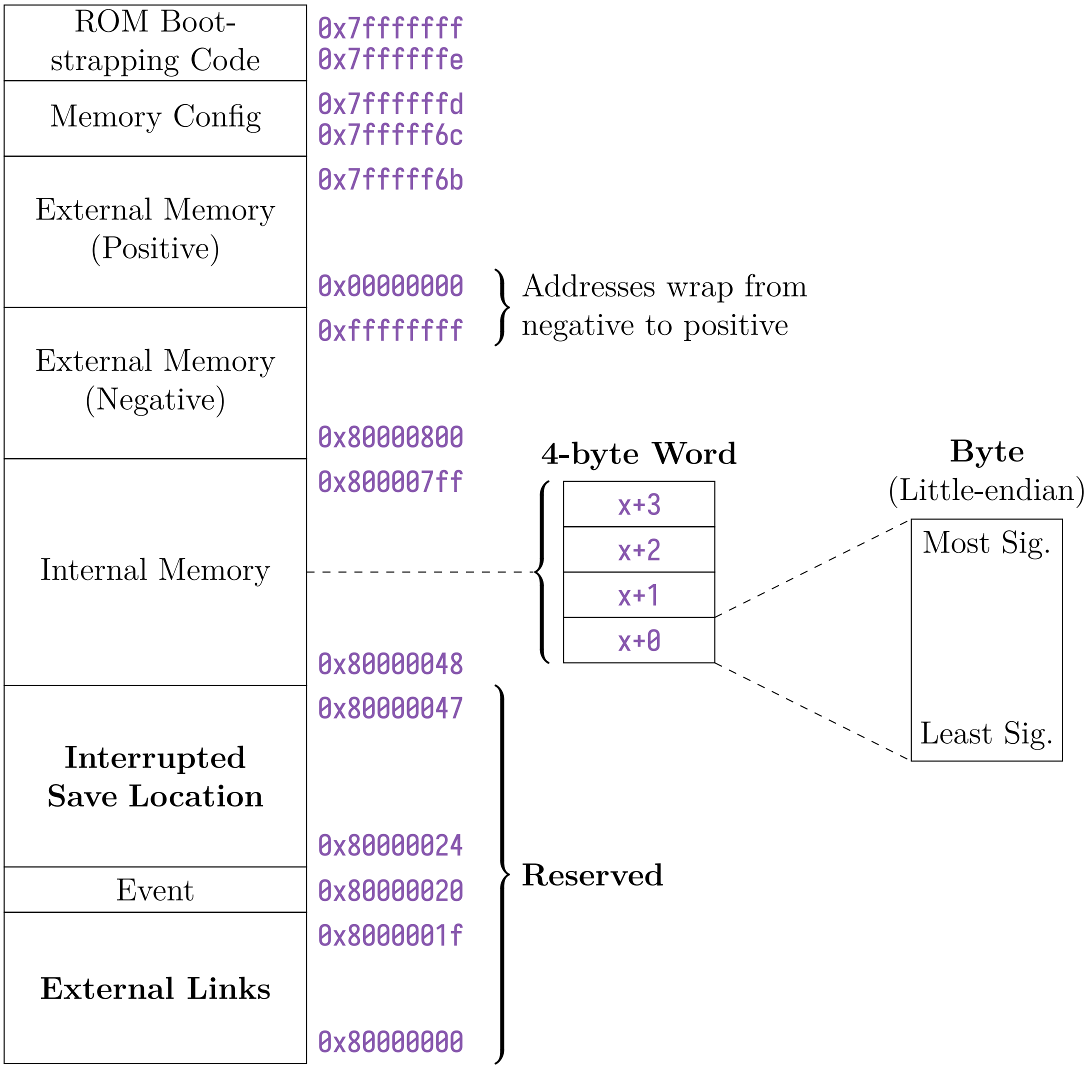


Figure 2: Transputer address space

Memory is divided into internal and external sections, with internal memory starting from the lowest address. Addresses are *signed*; they start at `0x80000000` and count up to `0xffffffff` before wrapping around to `0x00000000`. The address space is completely little-endian.

The spaces marked as **reserved** (Fig. 2) are not usually accessible directly, and are only modified indirectly through specific instructions for external communication or process interrupts; usable memory starts at `0x80000048`.

2.1.3 Machine Code

Transputer machine code is executed by the processor one byte at a time, with each byte containing one of 16 *core functions* (Table 2) in the upper nibble, and an operand between 0 and 15 in the lower nibble.

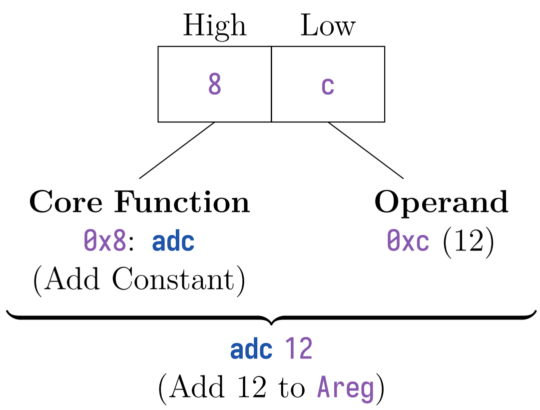


Figure 3: Single byte instruction

With each instruction only taking a single byte, the Transputer needs to employ additional techniques to accommodate operands outside of the limited range, and to expand the number of permissible operations beyond the core 16.

Func	Code	Name	Effect
j	0x0	Jump	Add operand to Iptr
ldlp	0x1	Load Local Pointer	Pushes Wptr + Oreg
pfix	0x2	Prefix	*Explained below*
ldnl	0x3	Load Non-Local	Pushes word from M[Areg + Oreg]
ldc	0x4	Load Constant	Pushes immediate value
ldnlp	0x5	Load Non-Local Pointer	Pushes pointer to M[Areg + Oreg]
nfix	0x6	Negative Prefix	*Explained below*
ldl	0x7	Load Local	Pushes word from M[Wptr + Oreg]
adc	0x8	Add Constant	Adds operand to Areg
call	0x9	Call	Procedure Call (See 2.1.4)
cj	0xa	Conditional Jump	Jump only if Areg = 0
ajw	0xb	Adjust Workspace	Offset Wptr by operand
eqc	0xc	Equals Constant	Push boolean value (Areg = Oreg)
stl	0xd	Store Local	Store Areg at M[Wptr + Oreg]
stnl	0xe	Store Non-Local	Store Areg at M[Breg + Oreg]
opr	0xf	Operate	*Explained below*

Table 2: Transputer Core Functions

To expand the domain of permissible operands, the pfix function is added before the main core function to indicate additional digits to the left. During execution, these additional digits are stored in Oreg. Figure 4 illustrates an example:

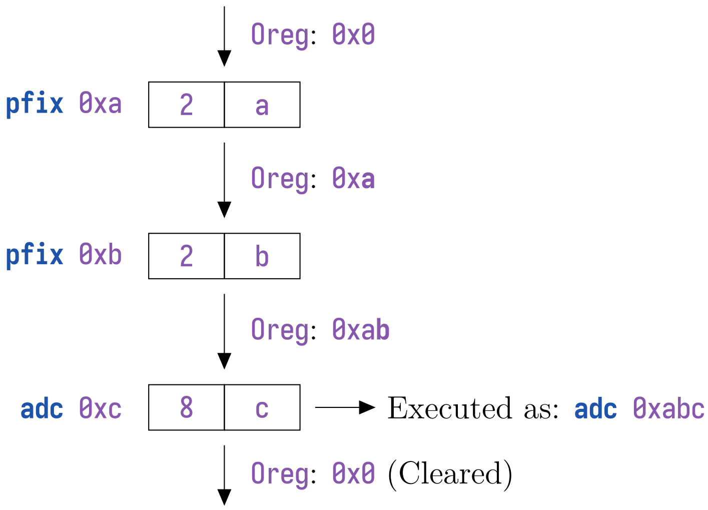


Figure 4: Prefix-extended operands

To deal with negative numbers, the `nfix` function is used. It inverts the value stored in `Oreg` and shifts it to the left by 4 bits, before adding a new nibble to the accumulated operand. Figure 5 illustrates such an example:

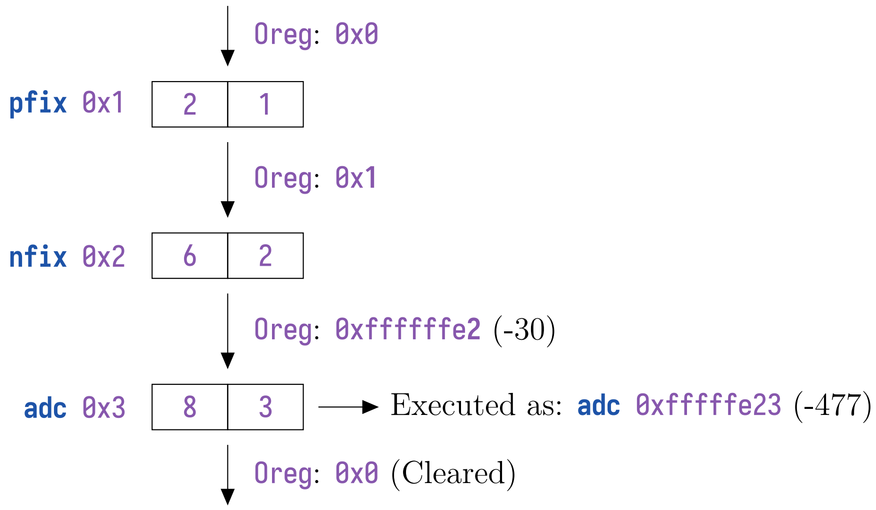


Figure 5: Negative operands

Beyond the core 16 functions, we can use the **`opr`** function, which selects an additional operation based on the operand. The T414 provides 87 additional operations which use values from the evaluation stack or workspace as their arguments.

2.1.4 Procedures

Within a sequential process, the Transputer provides mechanisms for calling procedures, in a similar manner to functions in other architectures.

Parameters are passed using the evaluation stack, with additional parameters placed into the workspace. Upon executing **`call`**, `Wptr` is moved 4 words down, and the evaluation stack and return address are copied into the newly allocated space. The procedure may choose to shift `Wptr` down further should it need more space (Fig. 6).

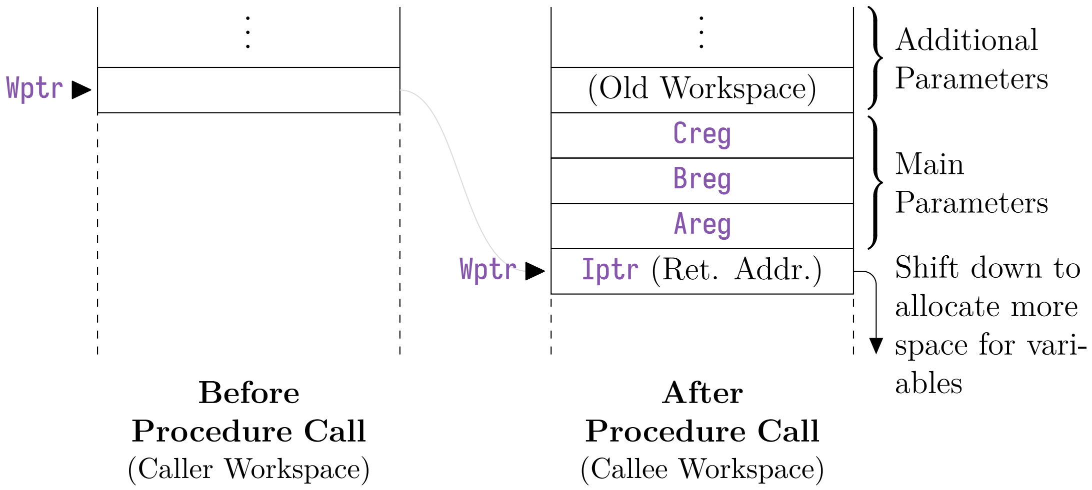


Figure 6: Workspace shifting during procedure calls

Since jump offsets are hardcoded in machine code, we need to use *stubs* to deal with dynamic procedure calling. When procedures are passed as arguments to other procedures, we call the stub procedure, which then uses the **gcall** operation to swap Areg and Ireg, essentially jumping to the supplied procedure (Fig. 7).

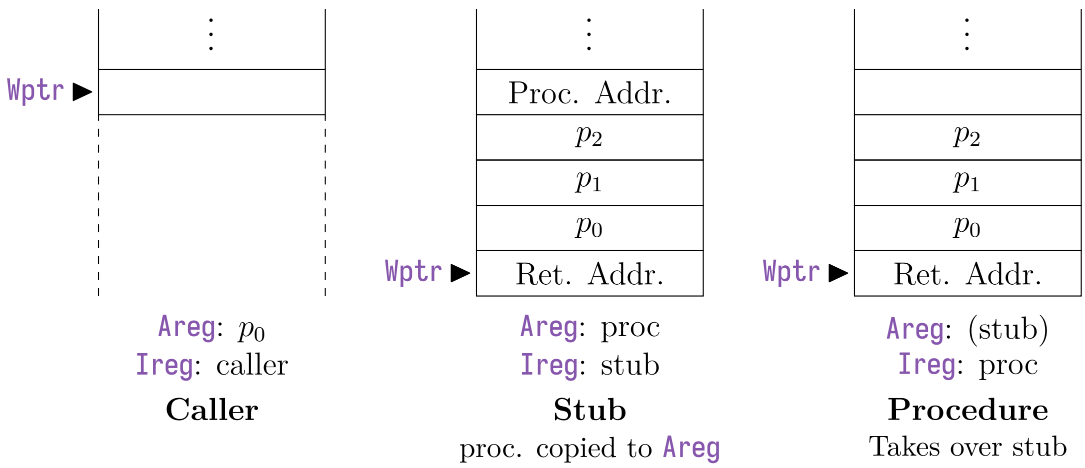


Figure 7: Calling procedures using a stub

2.2 Concurrent Processes

2.2.1 Process and Timer Queues

Processes are identified by their workspace addresses, which are always even, allowing us to use the final address bit to indicate priority. The top half of the workspace is used to store program variables, while the bottom half is reserved for specific purposes (Fig. 8).

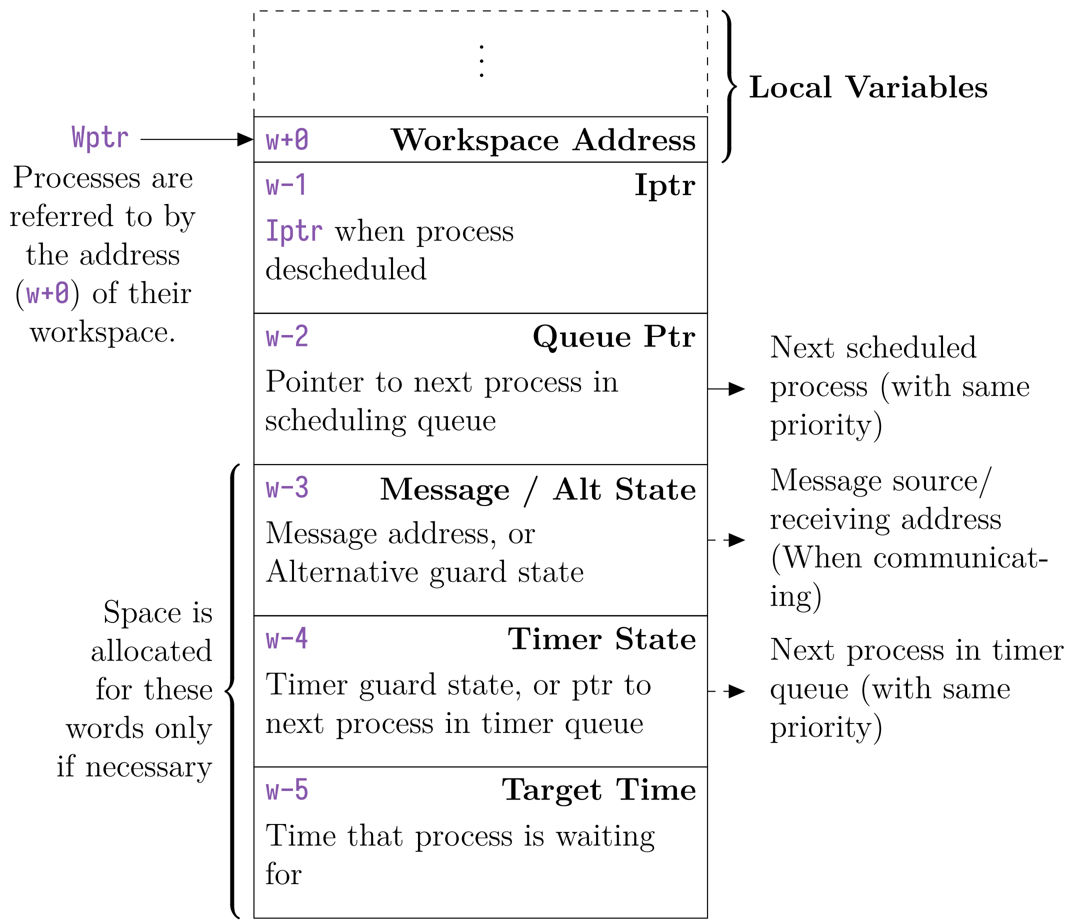


Figure 8: Process Workspace

Processes can either be high (`0`) or low (`1`) priority. Each priority has its own separate FIFO *timer queue* and *process queue*. *Active* processes are pushed to the back and executed at the front. Processes waiting for specific target times are placed into the timer queue.

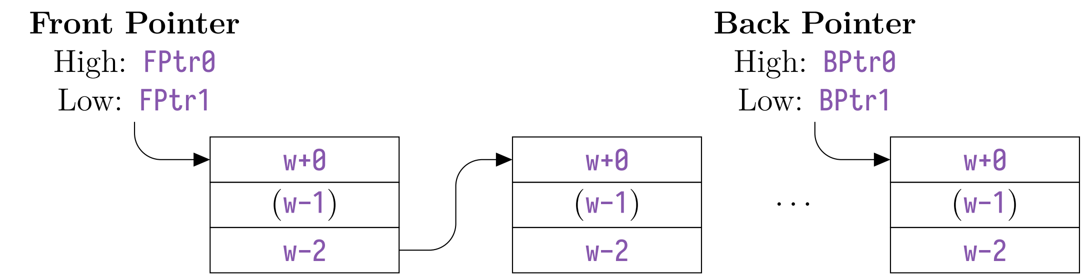
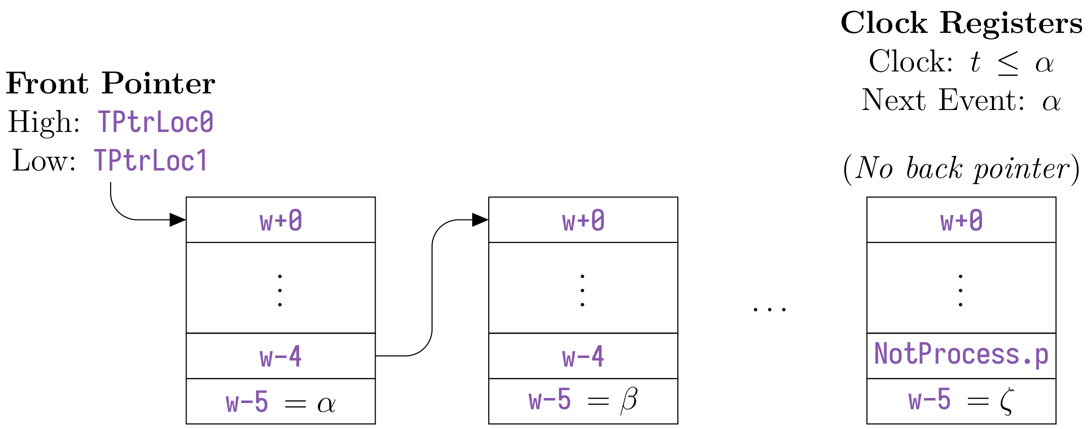


Figure 9: Process Queue



Condition: $\alpha \leq \beta \leq \cdots \leq \zeta$

Figure 10: Timer Queue

Process queues exist as linked lists in memory, with registers pointing to their first and last elements (Fig. 9). When process queues are empty, Head pointers are set to `NotProcess.p = 0x80000000` for empty queues.

Timer queues do not have back pointers; the last element points to `NotProcess.p`. Processes are arranged in ascending order based on their target time, as illustrated in Figure 10.

Two additional registers are used as clocks, with the high priority clock (`Clock0`) running 64 times as fast as the low priority clock (`Clock1`). There are also registers to indicate the earliest target time.

In all, there are 10 registers dedicated to process management (Table 3):

Register Type	**High**	**Low**
Process Queue Front Pointer	*FPtr0*	*FPtr1*
Process Queue Back Pointer	*FPtr0*	*BPtr1*
Process Clock	*Clock0*	*Clock1*
Next Event Time	*TNextReg0*	*TNextReg1*
Timer Queue Front Pointer	*TPtrLoc0*	*TPtrLoc1*

Table 3: Process Management Registers

2.2.2 Process Cycling and Timeslices

Transputers run instructions sequentially; low priority processes are cycled to simulate current active processes (Fig. 11):

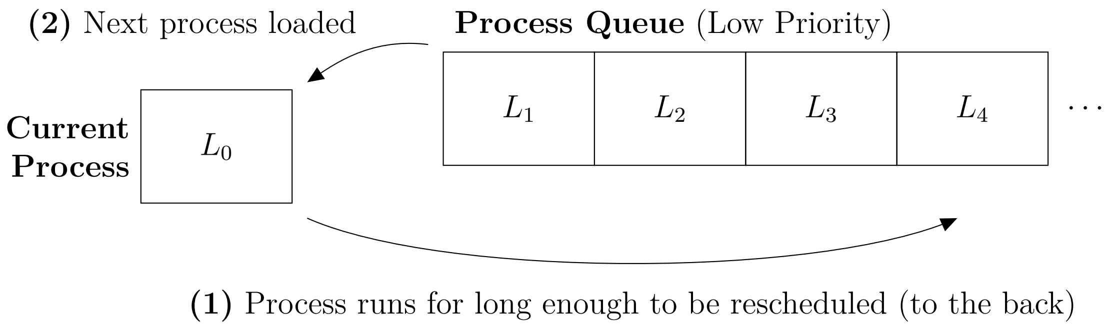


Figure 11: Cycling Low Priority Processes

Low priority processes are executed for 2 *timeslices* (equivalent to 5120 cycles) before they are rescheduled on the next *descheduling point instruction* (Table 4). These instructions signal to the processor that it is safe to switch to another process.

Name	Description	Name	Description
`in`	Input Message	`stoperr`	Stop on Error
`out`	Output Message	`altwt`	Alt Wait
`outbyte`	Output Byte	`j`	Jump
`outword`	Output Word	`lend`	Loop End
`taltwt`	Timer Alt Wait	`endp`	End Process
`tin`	Timer Input	`stopp`	Stop Process

Table 4: Descheduling Point Instructions

2.2.3 High Priority Process Interrupts

The high priority queue is always emptied before any low priority processes are allowed to execute, even if it means interrupting a low priority process. The register values of interrupted processes are temporarily stored in a fixed memory location (`0x80000024` to `0x80000047`).

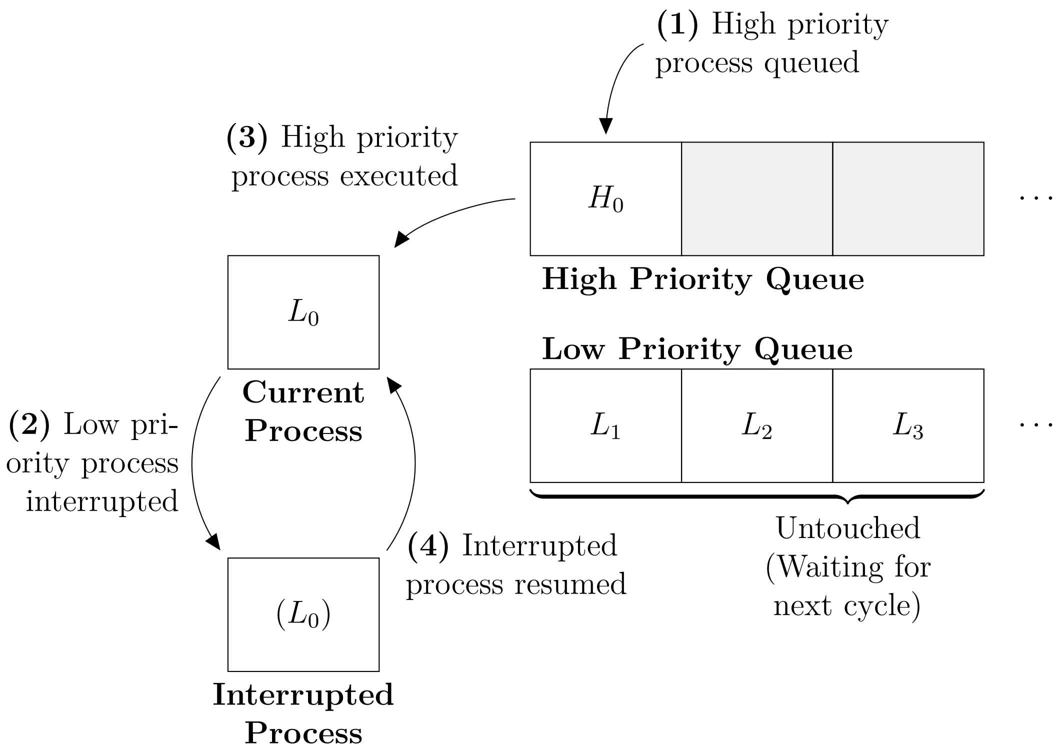


Figure 12: High Priority Process Interrupt

2.2.4 Internal Communications

Processes on the same processor can communicate between each other using internal channels. Internal channels take the form of a single word in memory, the *channel word*, which contains the channel status.

Channel words are first initialized to `NotProcess.p` to signify an inactive channel. The two connected processes can then communicate like so (Fig. 13):

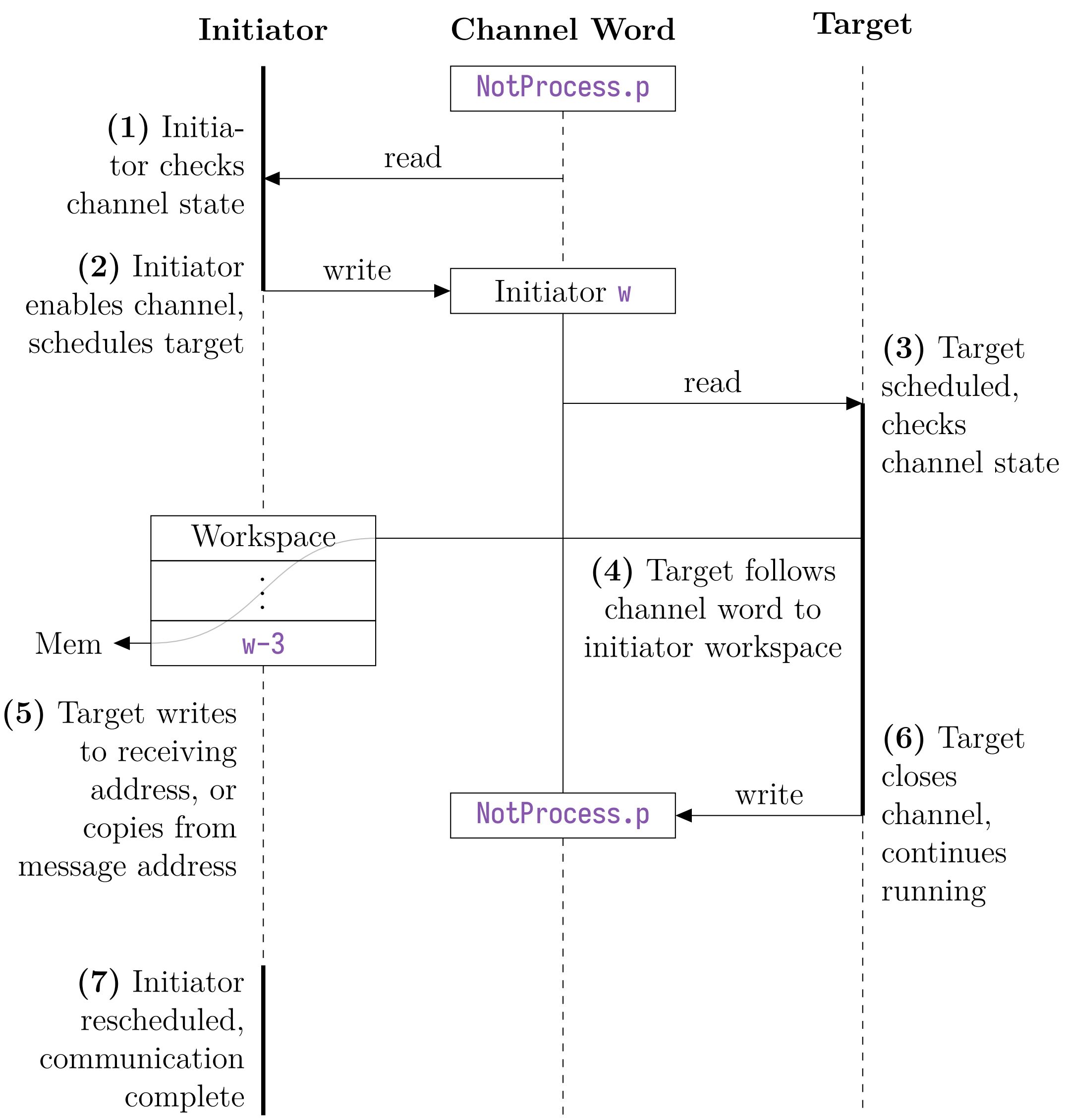


Figure 13: Channel communication

2.2.5 External Communications

Processes on different processors can also communicate via external physical links. Links contain their own registers, used to store message addresses and sizes. Communication occurs in the following order, with the two processes descheduled during message transfer (Fig. 14):

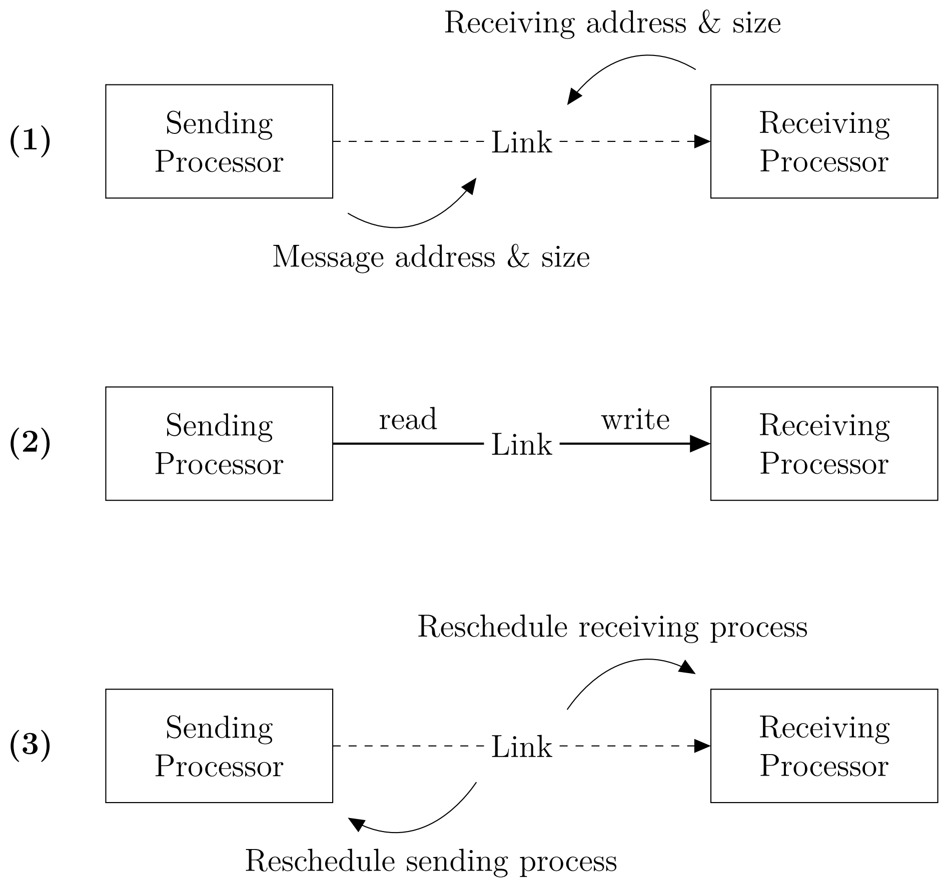


Figure 14: External Communication

2.2.6 Guarded Alternatives

Transputers also support guarded alternatives, where processes may branch into different paths depending on the conditions of *alternative guards*, of which there are three types: **Skip Guards**, **Channel Guards**, and **Timer Guards**. All three are associated with a boolean expression; channel guards are triggered by channel communications, and timer guards are triggered when the priority clock reaches a target time.

We can illustrate this with a CSP[1] process, using a new AFTER symbol for indicating reaching a specific target time on the priority timer.

Take, for example, the following CSP process P, where branches B_0, B_1, and B_2 are placed behind a skip, channel, and timer guard respectively:

$$
\begin{aligned}
P = {} & e_0 \,\&\, B_0 \\
& \square\, e_1 \,\&\, (c?v \rightarrow B_1) \\
& \square\, e_2 \,\&\, (\textsc{After}\ t \rightarrow B_2)
\end{aligned}
$$

[1]Communicating Sequential Processes, based on the work of Tony Hoare [9] [10]

The three guards are first *enabled* in order, before the process is descheduled during the *alt wait* (Fig. 15). Skip guards allow us to *skip* the wait if the necessary conditions are fulfilled.

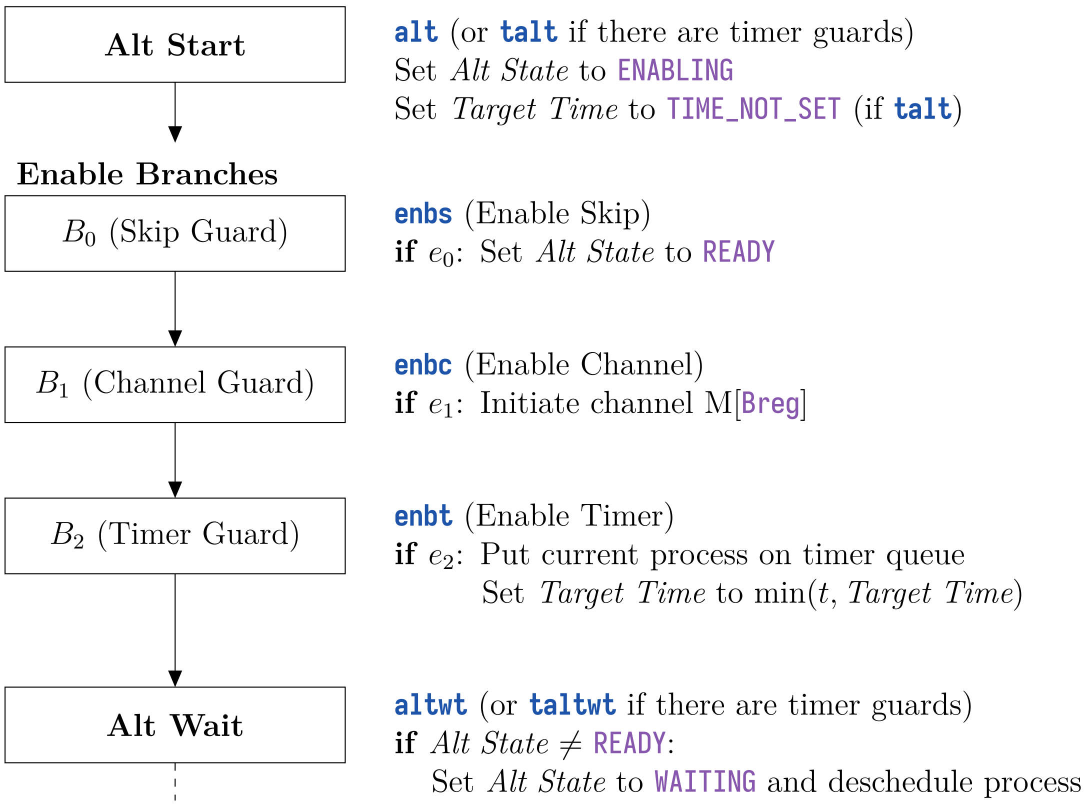


Figure 15: Enabling Branches

After the wait, each branch is *disabled*, where they check if they have been chosen (Fig. 16). The process then jumps to the chosen branch.

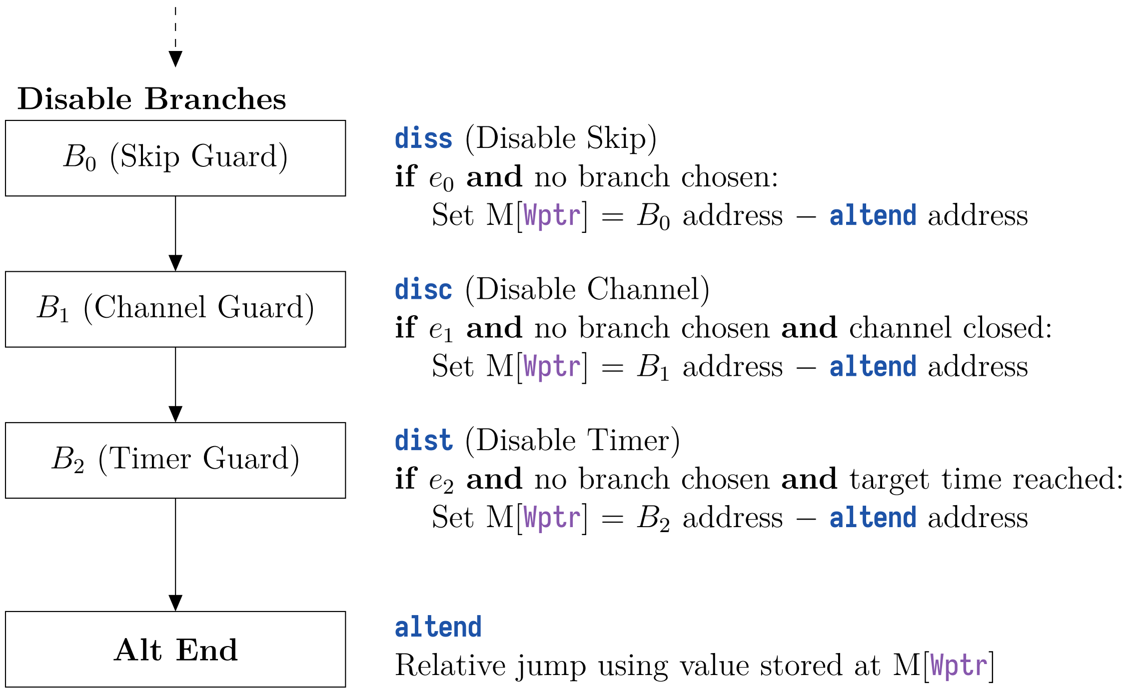


Figure 16: Disabling Branches

3 Transputer Assembler and Multi-processor Simulator

The program that has been written to simulate the T414 instruction set is called the Transputer Assembler and Multi-processor Simulator (TAMS). It is a console program implemented in C++20 and consists of a command-based interface and separate utility modules, as illustrated in Figure 17.

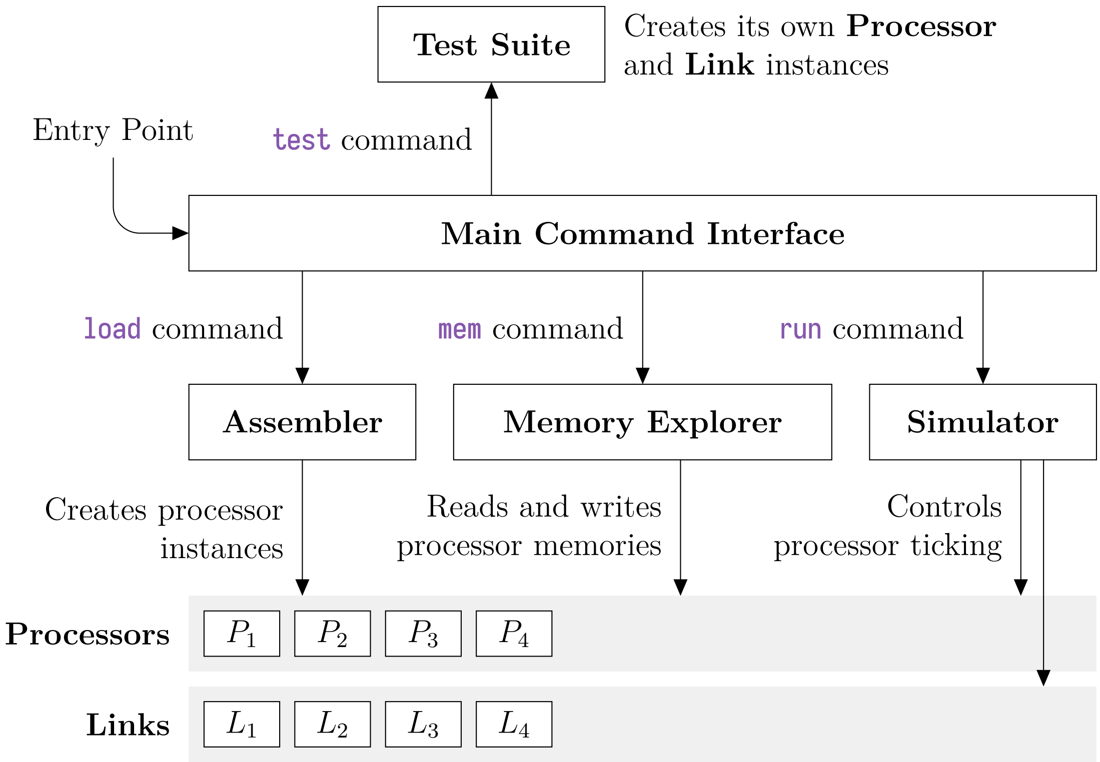


Figure 17: TAMS Program Structure

3.1 Standardized Transputer Assembly

Existing publications generally use nonstandardised, assembler-independent *pseudo-assembly* [7] [11] that do not contain contain the necessary features we need for an assembler; we will thus have to create a new standard.

3.1.1 Assembly Syntax

We first start by examining examples of pseudo-assembly in ACWG, such as the following:

Snippet 1 Pseudo-Assembly Example on Concurrent Process Initialization

```
    ldc 3; stl 1;        # Immediate value operands
    ldc L5 - L6; ldpi;   # Label offset operands
L6: stl 0;
    ldc L1 - L2; ldlp WP;
    startp;
L2: ldc L3 - L4; ldlp WQ;
    startp;
L4: R; ldlp 0; endp;    # R, P, Q are abstracted sequences of processes,
L1: P; ldlp -WP; endp;
L3: Q; ldlp -WQ; endp;  # Label operands
L5:
```

We can identify all three formats of operands from this example: immediate values, labels, and label offsets.

3.1.2 Operand Types

The same operand format can be interpreted differently depending on the instruction. For instance, the jump instruction (`j`) shifts `Iptr` in bytes, while the adjust workspace (`ajw`) instruction shifts `Wptr` in words. There are three operand types, summarised in Table 5. Note that Addr[x] refers to the byte address of x, while Addr[Next] refers to the byte address of the following instruction.

Operand Type	**Imm.** n	**Label** *b*	**Difference** *a* - *b*
Raw Value	n	Addr[*b*]	Addr[*a*] - Addr[*b*]
Offset (Bytes)	n	Addr[*b*] - Addr[Next]	Addr[*a*] - Addr[*b*]
Offset (Words)	n	-	(Addr[*a*] - Addr[*b*]) / 4

Table 5: Operand Type Interpretations

When a byte offset operand type is supplied with a single label, we subtract the address of the next instruction. This allows us to write jump instructions with just single labels.

We can classify all core functions based on their operand type (Table 6):

Function	Name	Operand Type
`j`	Jump	Offset (Bytes)
`ldlp`	Load Local Pointer	Offset (Words)
`pfix`	Prefix	-
`ldnl`	Load Non-Local	Offset (Words)
`ldc`	Load Constant	Raw Value
`ldnlp`	Load Non-Local Pointer	Offset (Words)
`nfix`	Negative Prefix	-
`ldl`	Load Local	Offset (Words)
`adc`	Add Constant	Raw Value
`call`	Call	Offset (Bytes)
`cj`	Conditional Jump	Offset (Bytes)
`ajw`	Adjust Workspace	Offset (Words)
`eqc`	Equals Constant	Raw Value
`stl`	Store Local	Offset (Words)
`stnl`	Store Non-Local	Offset (Words)
`opr`	Operate	Raw Value

Table 6: Core Function Operand Types

3.1.3 Operations & Macros

As explored in 2.1.3, we use `opr` to select from a list of additional operation. We can write those additional operations in *mnemonic form* in assembly, but they would need to be expanded when generating machine code (Fig. 18).

Figure 18: Mnemonic Expansion

When the operation is selected using a value larger than `0xf`, the assembler would have to expand it further into prefix form (Fig. 19).

Figure 19: Operands outside of range

3.1.4 Assembler Directives

We also have to add a few additional assembler directives to indicate initial processor conditions, and code/workspace locations. Table 7 summarises these directives:

Directive	Type	Purpose
`%istart` a	Preamble	Initial `Iptr` value
`%wstart` a	Preamble	Initial `Wptr` value
`.addr` a	Instruction	Skip to a specific address
`.break`	Instruction	Debug breakpoint
`.zero` n	Instruction	Insert empty bytes
`.byte` b	Instruction	Insert raw byte data
`.word` w	Instruction	Insert raw word data

Table 7: Operand Type Interpretations

Preamble-type directives can only be written at the top of the assembly file, while *Instruction*-type directives are parsed like instructions and can be inserted into code.

The syntax is as follows:

Snippet 2 Demonstration of Assembler Directives

```
%istart 0x80000060       # Iptr starts at 0x80000060
%wstart 0x80000204       # Wptr starts at 0x80000204

.addr 0x80000060         # Start writing the following code at 0x80000060
main:
    ldl 0
    .break               # Breakpoint for debugging
    adc 1
    .byte 0xff           # Write the raw byte 0xff

.addr 0x80000200         # Start writing the following code at 0x80000200
    .zero 4              # Insert 4 bytes of 0x00
    .word 0x00000001     # Insert raw word 0x00000001
```

3.2 Assembler

The **Assembler** module in TAMS accessible via the `load` command in the main command interface. It converts programs written in assembly into machine code (Fig. 20):

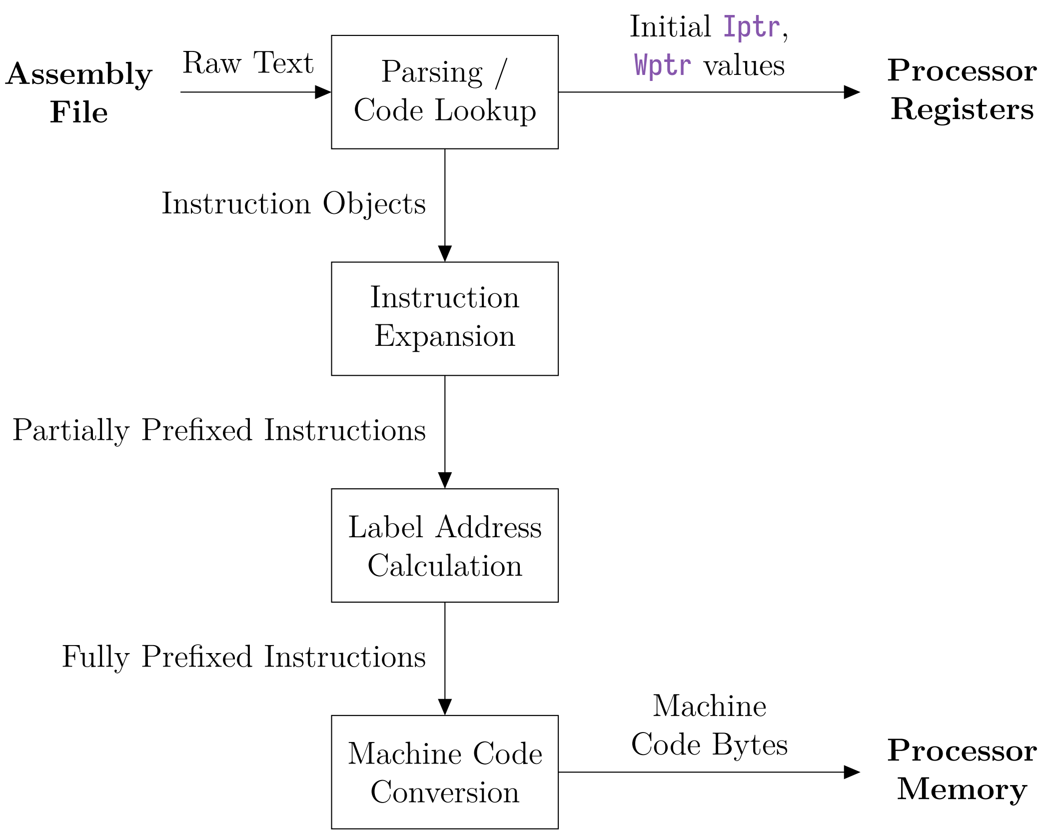


Figure 20: Assembler Structure

Assembling programs first start with parsing and code lookup, where text is converted into *Instruction Objects* (Struct instances) using a lookup table, defined as such:

Snippet 3 Instruction Struct (`assembler.h`)

```
struct Instruction {
    bool specialOp  = false;    // Is a directive
    uint32_t code   = 0;        // Function (instruction) as byte code
    int bytes       = 0;        // Estimated byte size after expansion
    int selfLabel   = -1;       // Label ID of current instruction, if any
    bool hasOperand = false;    // Instruction has operand
    int operand     = 0;        // Candidate operand
    int labelTarget = -1;       // Jump to label (using label ID)
    int labelRef    = -1;       // Jump counted relative to this label
};
```

The initial pointer register values are also read and saved at this point; instructions in mnemonic form have already been prefixed in the lookup table. They are then sent through a series of steps to evaluate their operands, before conversion to machine code.

3.2.1 Instruction Expansion

Instruction expansion is done in multiple stages, starting with instructions that do not contain labels (Fig. 21):

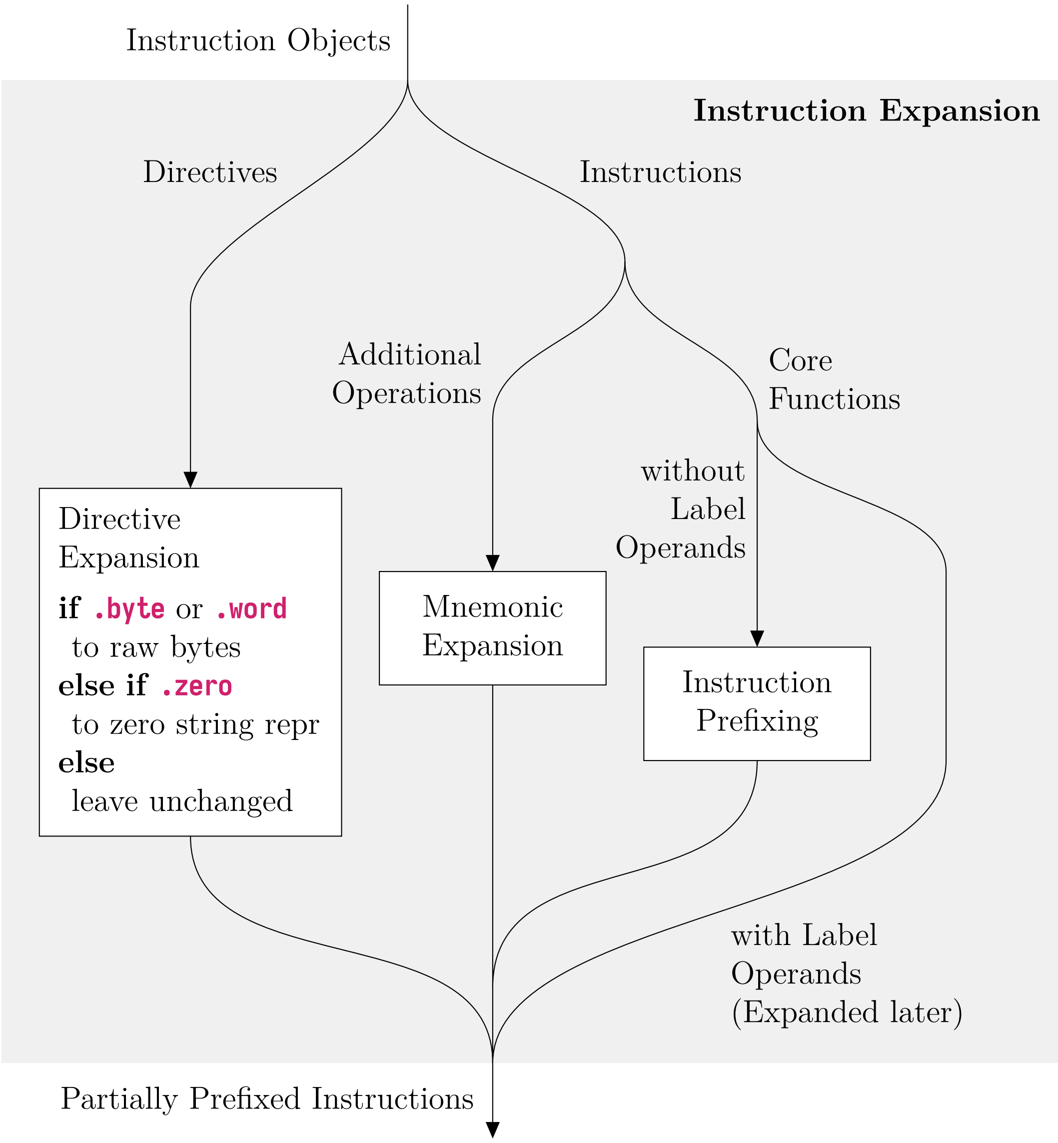


Figure 21: Instruction Expansion Step

Of all the assembler directives, `.byte` and `.word` are immediately expanded into raw bytes; `.addr` and `.break` can only be expanded after label addresses are finalised.

`.zero` instructions are saved as instruction objects like so:

Snippet 4 Intermediate Representation of `.zero` n

```
Instruction {
    specialOp   = true;
    code        = 0xffff04;
    bytes       = n;        // Estimated byte size
    hasOperand  = false;
    operand     = n;        // Candidate operand
    // Other fields inherited from parsed instruction object
};
```

Operation codes are fully expanded using a lookup table. Core functions are prefixed based on their operand, using the algorithm below, where `f` is the function to be prefixed, and $\sim x$ represents a bitwise *not*:

Algorithm 1 Prefixing constants

```
1: function PREFIX(f, x)
2:     if 0 ≤ x < 16 then
3:         return f(x)
4:     else if x ≥ 16 then
5:         return PREFIX(pfix, x >> 4); f(x & 0xF)
6:     else if x < 0 then
7:         return PREFIX(nfix, ~x); f(x & 0xF)
```

3.2.2 Label Address Calculation

As the length of a prefixed instruction is determined by the magnitude of its operand, the size of two expanded instructions can be mutually dependent (Fig. 22):

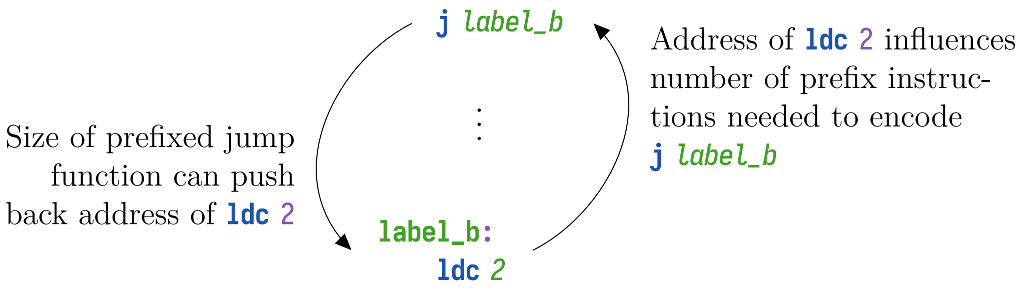


Figure 22: Label Address Dependencies

This prevents us from easily expanding instructions with labels in a single pass. We thus adapt the following iterative refinement process provided by ACWG, modified to work with directives:

Algorithm 2 Iterative Label Address Calculation

Initialize map **M** : label $\rightarrow$ address
Assign *fixed size estimates* to instructions with known sizes
Assign *0* to instructions with label operands
while Estimates changed **do**
    Update label addresses in **M** based on estimates
    Calculate values of operands in the form `a - b`
    Update estimates based on new operand values

To determine the size of all instructions with label operands, we use the following algorithm:

Algorithm 3 Finding prefixed instruction size based on operand value

```
function PrefixSize(operand)
    if 0 ≤ operand < 16 then
        return 1
    let est = 1
    let temp = operand
    while temp ∉ [0, 15] do
        est = est + 1
        if temp < 0 then temp = ~temp
        temp = temp » 4
    return est
```

Lastly, `.zero` and `.addr` are expanded into full strings of zeroes of the required lengths.

3.3 Processor Simulation

To support multiple processors running simultaneously, processors are defined as a class, and instances of processors can be spawned by loading multiple assembly programs.

3.3.1 Processor Stepping

With multiple processes stepped together externally by the `Simulator` class, processor instances have to provide a step function to simulate the passing of a single clock cycle. They store their internal states, updated on each step function call (Fig. 23):

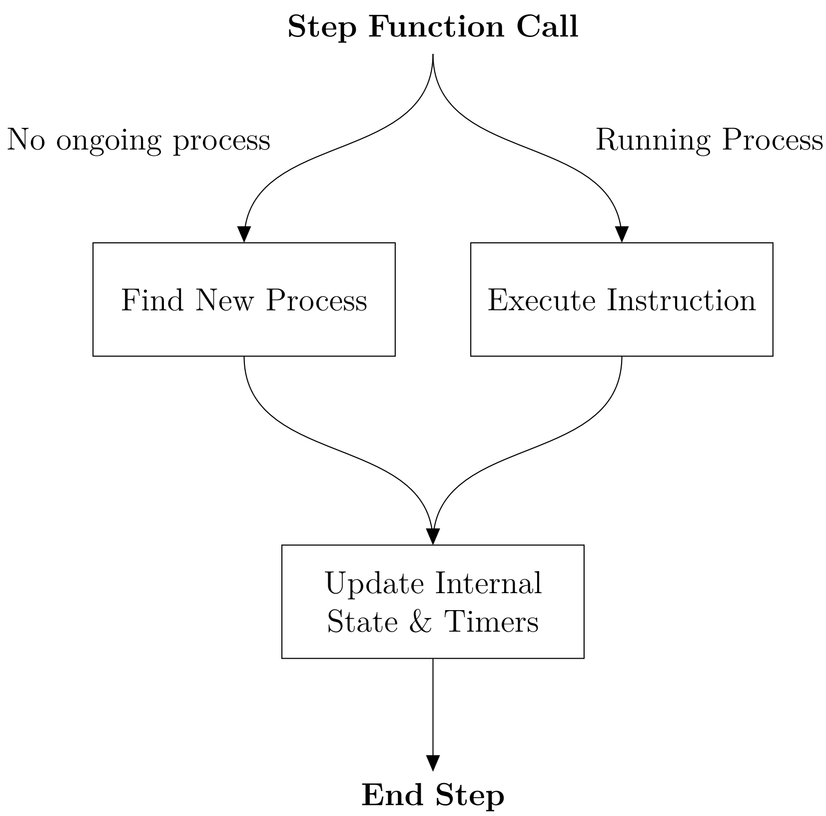


Figure 23: Processor Step Flow

TAMS provides two ways of stepping through simulations: the *step* command, where all processors execute a single cycle, and the *next* command, which skips to the next breakpoint, unless interrupted by an error flag.

3.3.2 Timer Updates

To facilitate the timekeeping, we also introduce a set of helper counters and flags:

Name	Name in Code	Description
Running Timers	`runningTimers`	Whether timers are running (Boolean)
Cycle Count	`cycles`	Total cycles executed
Timeslice Cycles	`tsCycles`	Cycles elapsed in current timeslice
Process Timeslices	`timeslice`	Timeslices elapsed in current process
Low Clock μs	`loClockUs`	μs elapsed in current low clock interval
μs Cycles	`usCycles`	Cycles elapsed in current μs interval
Delayed Cycles	`pastCycles`	Cycles since last executed instruction
Buffered Cycles	`bufferedCycles`	Cycles needed for current instruction

Table 8: Timekeeping Variables

This allows us to coordinate all the timers like so (Fig. 24):

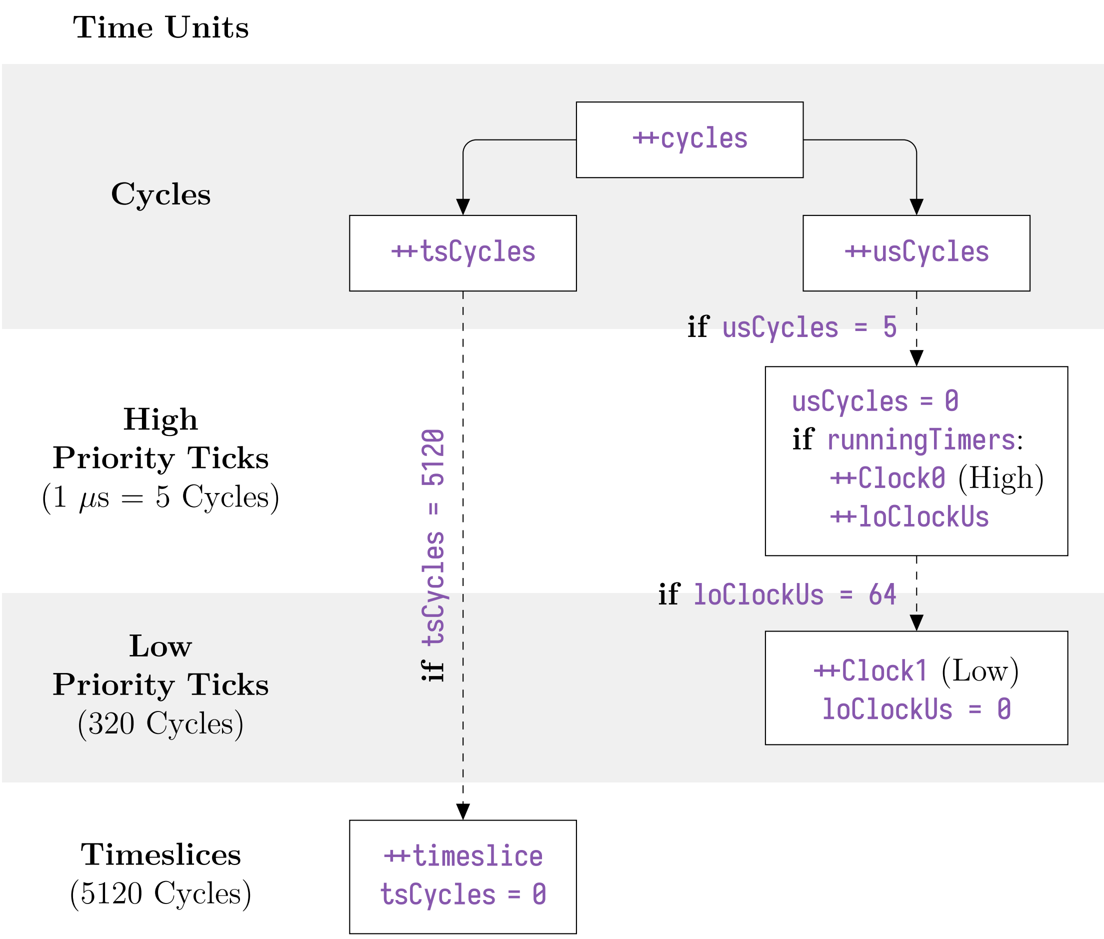


Figure 24: Timer Update Flow

3.3.3 Instruction Execution

As different instructions take different numbers of cycles to execute, we have to predict execution length and buffer the effects of instructions to simulate their behaviour (Fig. 25):

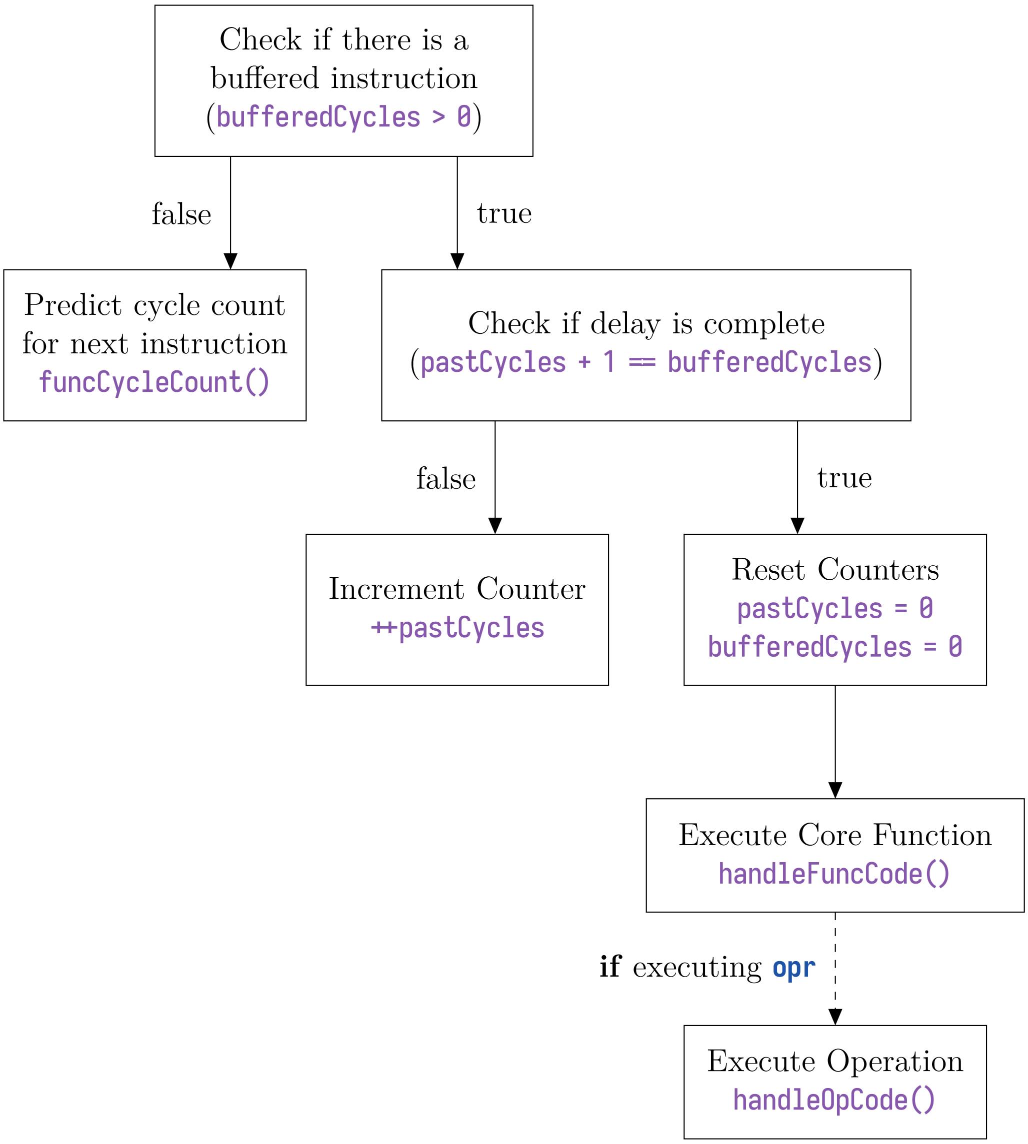


Figure 25: Instruction Execution Flow

TAMS only executes buffered instructions after the delay such that any changes to registers or memory only happen after the full execution time period has elapsed.

3.3.4 Internal Process Management

To support the process scheduling, additional helper functions and flags have been added to the `Processor` class:

- **Descheduling Check Flag** (`bool descheduleCheck`), indicates when it is safe to perform a descheduling check to cycle low priority processes;
- **Schedule Function** (`Processor::schedule`), called when a process becomes active and needs to be queued;
- **Timer Queue Function** (`Processor::queueTimer`), called when a process starts waiting for a target time;
- **Starting Process Flag** (`bool startingProc`), used to indicate that no process is running and the next process should start.

These functions do not interfere with the current running process; they set up the internal processor state for the next cycle (Fig. 26):

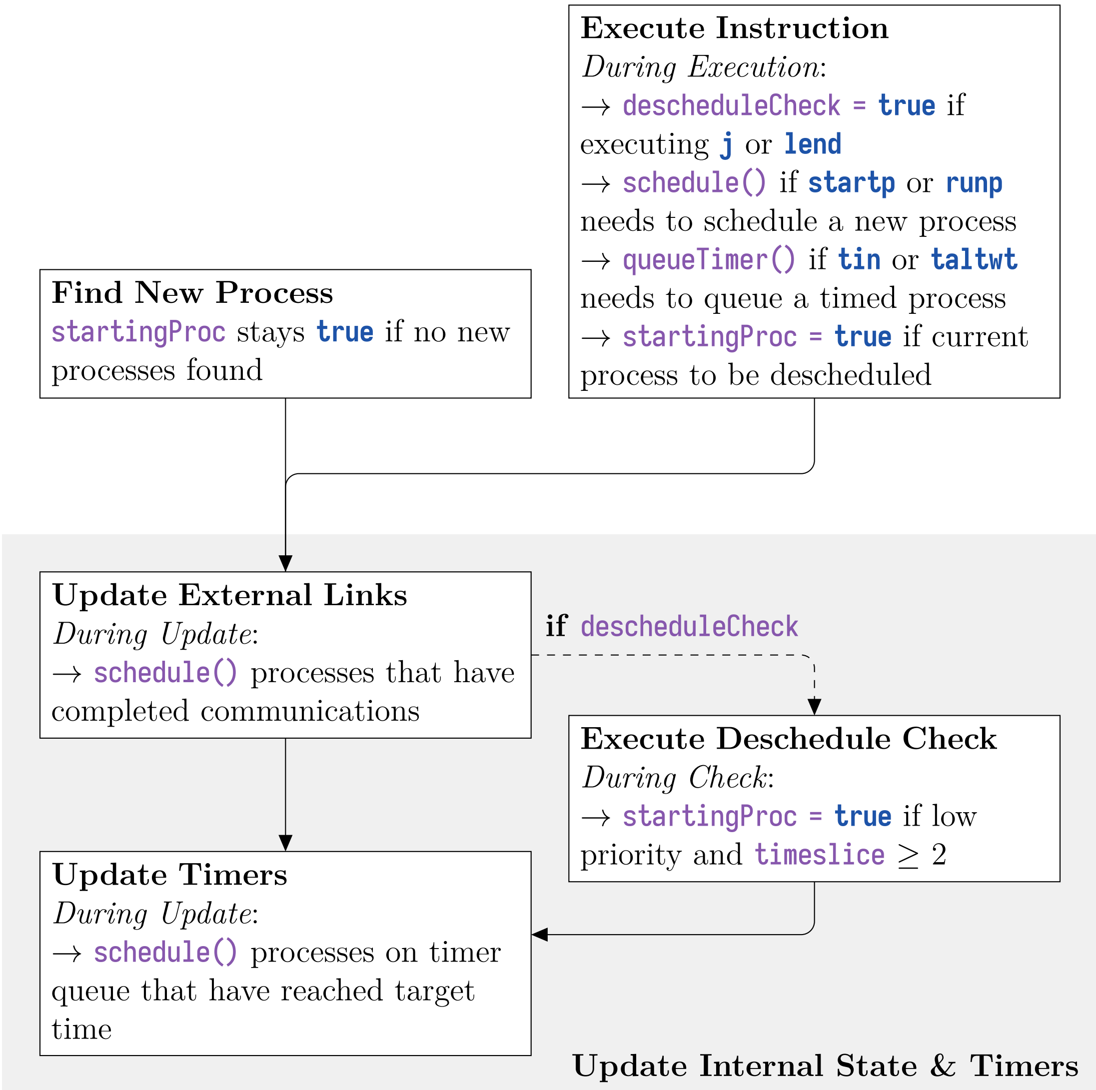


Figure 26: Process Management Flow

Notably, both descheduling points and process starts are implemented as boolean flags rather than functions, so that we can delay the corresponding state changes to after instruction execution.

3.4 External Communication

To facilitate external communication between multiple processors, physical links have been abstracted within TAMS using the `Link` class.

3.4.1 Links & Channels

In TAMS, link instances connect two processors with two channels like so (Fig. 27):

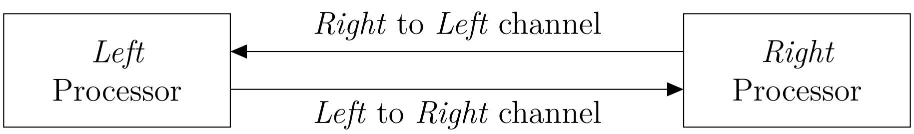


Figure 27: Link Channels in TAMS

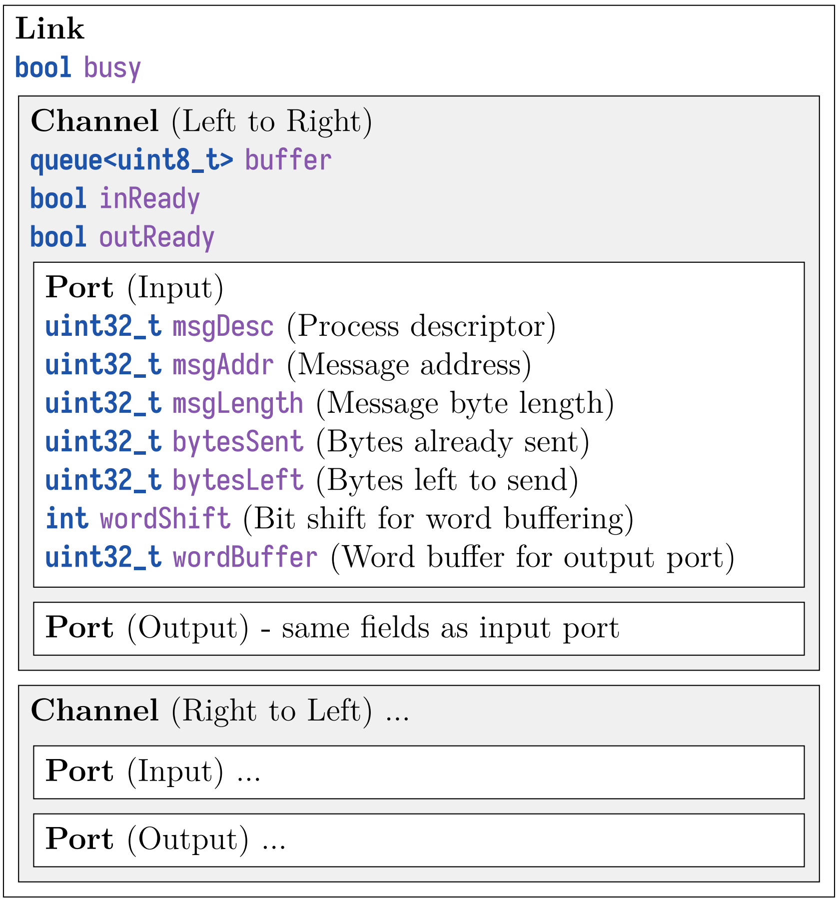


Figure 28: Link Structure
3.4.1

To facilitate communication, each link contains a number of variables; this includes registers that were already present on the original hardware, and additional helper variables. This is summarised above in Figure .

Since alternatives require us to synchronise the two sides before the receiver calls **`in`**, we have to use additional readiness indicators (Fig. 29):

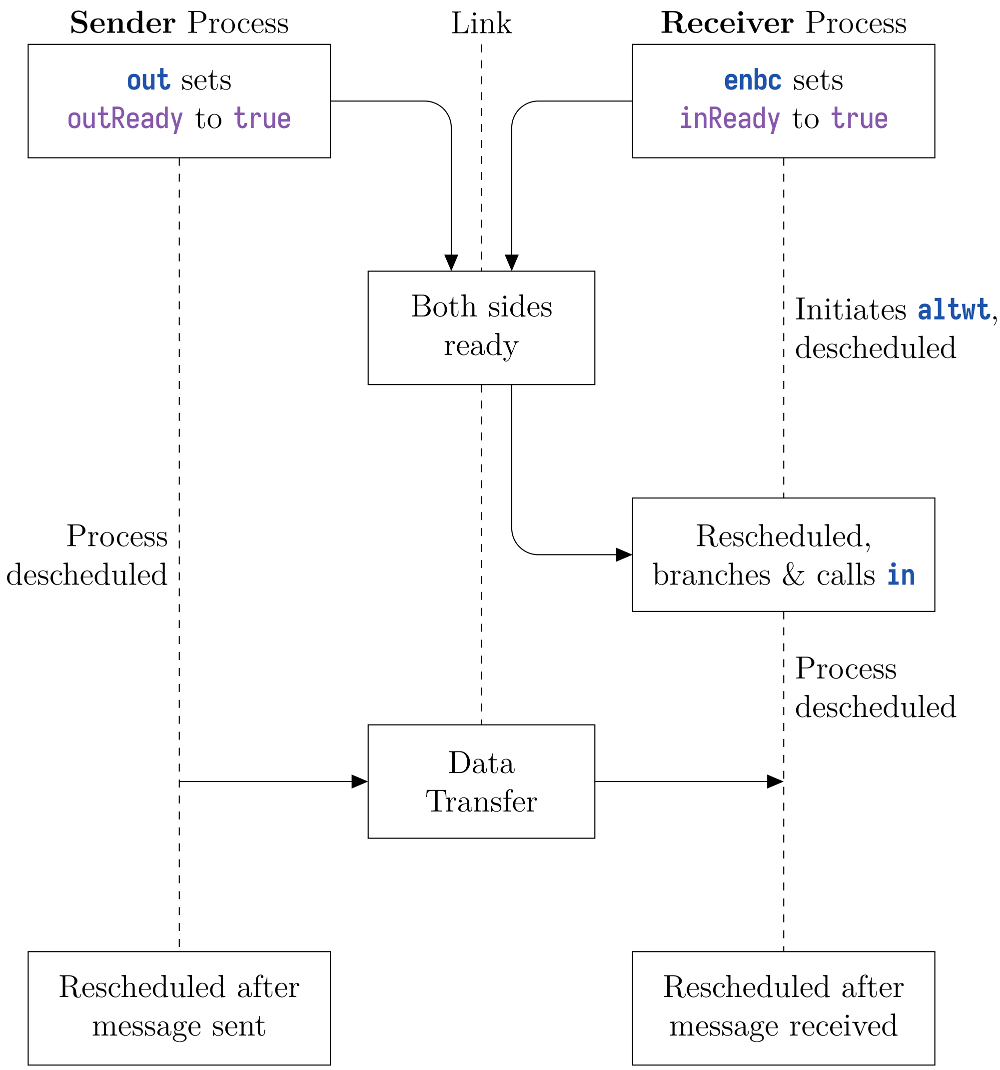


Figure 29: Alternative Guard Synchronisation

To send messages, channels alternate between two states depending on the buffer, which holds one byte (Fig. 30):

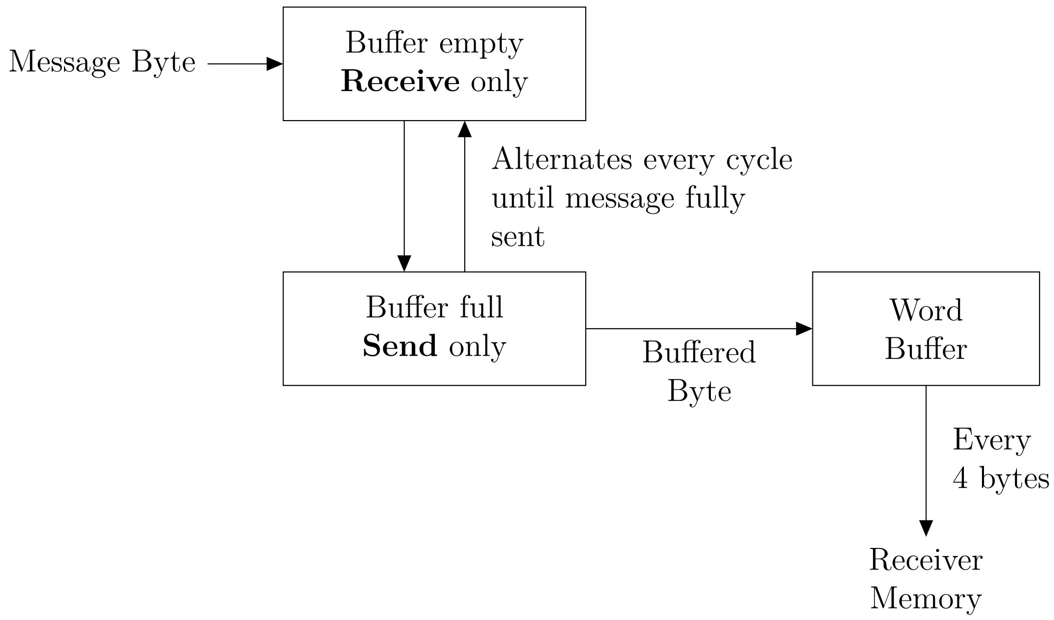


Figure 30: Link Data Transfer

3.5 Debugging Tools

TAMS contains two sets of tools for debugging programs - a memory explorer for checking values in memory, and a testing suite for automated testing of programs in bulk.

3.5.1 Memory Explorer

The memory explorer is accessed through the `mem` command and provides its own command interface to navigate between different pages of memory. Since processor instances provide public functions for querying the values of bytes in memory, the memory explorer simply retrieves the values it needs that way.

Snippet 5 Memory Explorer Interface

```
MEMORY EXPLORER

               0      1      2      3      4      5      6      7      8      9      A      B     ..
0x80000000 |[00   ] 00     00     00     00     00     00     00     00     00     00     00     ..
           |[j    ] j      j      j      j      j      j      j      j      j      j      j      ..
0x80000010 | 00     00     00     00     00     00     00     00     00     00     00     00     ..
           | j      j      j      j      j      j      j      j      j      j      j      j      ..
0x80000020 | 00     00     00     00     00     00     00     00     00     00     00     00     ..
           | j      j      j      j      j      j      j      j      j      j      j      j      ..
0x80000030 | 00     00     00     00     00     00     00     00     00     00     00     00     ..
           | j      j      j      j      j      j      j      j      j      j      j      j      ..
0x80000040 | 00     00     00     00     00     00     00     00     00     00     00     00     ..
           | j      j      j      j      j      j      j      j      j      j      j      j      ..
0x80000050 | 00     00     00     00     00     00     00     00     00     00     00     00     ..
           | j      j      j      j      j      j      j      j      j      j      j      j      ..
0x80000060 | 27 '   2f /   2f /   2f /   2f /   2f /   6f o   40 @   d0     43 C   d1     70 p   ..
           | pfix   pfix   pfix   pfix   pfix   pfix   nfix   ldc    stl    ldc    stl    ldl    ..
0x80000070 | 00     00     00     00     00     00     00     00     00     00     00     00     ..
           | j      j      j      j      j      j      j      j      j      j      j      j      ..

Instruction Pointer: 0x80000060
Workspace Pointer:   0x80000200

(80000000: 00) mem>
```

Using the tool, we can navigate and edit bytes in memory using a command interface. This is facilitated by a cursor (indicated by `[]` brackets). Each byte within memory is also translated into its relevant ASCII symbol and function mnemonic where appropriate.

One intentional design choice is to keep the processor stepping tool (the simulator) and the memory explorer separate; since memory space is typically very large, the simulator only logs changes to specific memory addresses, with any tasks involving viewing the full memory space relegated to the memory explorer.

3.5.2 Test Suite

To systematically test features in the simulator to ensure that programs reliably produce their expected output, TAMS also includes a separate testing suite that automatically iterates through a list of predefined tests, each configured to create its own set of processor and link instances, with checks on its final state after the processors halt.

This is defined using a custom `.tamst` filetype:

Snippet 6 TAMS Test Configuration

```
proc ./programs/extsend.tams send
proc ./programs/extrecv.tams recv

link send 0 recv 0

test send Areg 0
test recv Areg 4
test send Wptr 0x80000204
test send Iptr 0x80000069
test send flag 0
test recv 0x80000204 3
```

All tests defined within the test directory will be run when the `test` command is executed. Each test creates an independent environment where processors and links are created from scratch using the `proc` and `link` functions. After running and hitting a breakpoint, or halting, checks are run using the `test` function, before the environment is deleted in preparation for the next test.

Checks can be done on registers (`test send Areg 0`), specific memory locations (`test recv 0x80000204 3`), and error flags (`test send flag 0`) on each processor.

4 Evaluation of Example Programs

To fully explore the unique instruction set and architecture of the Transputer, it helps to reimplement familiar programs within the instruction set so that we may fully grasp its limitations and advantages. In particular, the stack machine design and the built-in process management lend themselves to rather interesting approaches when writing with the instruction set. We will first explore sequential programs, before moving on to concurrent ones.

4.1 Using TAMS

After starting up TAMS, users are greeted with the main command interface, from which they are able to access top-level commands. These are listed in the help page:

Snippet 7 TAMS Startup Message & Help

```
Transputer Assembler & Multiprocessor Simulator v0.4.0
Type 'help' for a list of commands.
tams> help
TAMS Command Help:
  clear - Clear all processor instances
  create [Memory size] [Processor name] - Create new empty processor instance
  exit - Exit TAMS
  help - List all commands
  link <Left Proc> <Left Port> <Right Proc> <Right Port> - Create a link between two processors
  list - List all processor instances
  load <File> [Processor name] - Load an assembly file into a new processor instance
  mem [Processor index] - Explore the memory of a processor instance
  run - Open the simulator interface for running processors
```

To load and run a program, we must first load the TAMS file containing the assembly code using the `load` command. This creates a new processor instance and automatically assembles the code, which is written directly into the processor.

The `load` command can be done as many times as required to create multiple processors, and the `list` command displays all created processors:

Snippet 8 Loading programs on TAMS

```
tams> load ./programs/extsend.tams send
Loaded program with no warnings.
tams> load ./programs/extrecv.tams recv
Loaded program with no warnings.
tams> list
Processor instances:
Index   Name
0       send
1       recv
```

We can then create any necessary external links using the `link` command, specifying which ports to use for both processors:

Snippet 9 Creating external links on TAMS

```
tams> link send 0 recv 0
Created link with no warnings.
```

To start the simulation, we can use the `run` command, which brings us into the simulator interface, where we can either use the `step` command to advance all processors by one clock cycle, or `next` command to skip to the next breakpoint or halt. The following snippet shows the output from running `step` once, with both processors executing a single byte.

Snippet 10 Running simulations on TAMS

```
tams> run
send> Ran func pfix with nibble 7 at address 0x80000060
recv> Ran func pfix with nibble 7 at address 0x80000060

send:
  | A:  0x00000000 | B:  0x00000000 | C:  0x00000000 | W:  0x80000200 | I:  0x80000061 | O:  0x00000070 | TS: 1 (0), Cycles: 1
  | Hi: 0x00000000 | Lo: 0x00000000 | Break:      No | Priority:  Low | Error:      No | Clock:      No | StartProc:  No | Intr: 00000000
  | hiQueue: None
  | loQueue: None
  | hiTimerQueue: None
  | loTimerQueue: None

recv:
  | A:  0x00000000 | B:  0x00000000 | C:  0x00000000 | W:  0x80000200 | I:  0x80000061 | O:  0x00000070 | TS: 1 (0), Cycles: 1
  | Hi: 0x00000000 | Lo: 0x00000000 | Break:      No | Priority:  Low | Error:      No | Clock:      No | StartProc:  No | Intr: 00000000
  | hiQueue: None
  | loQueue: None
  | hiTimerQueue: None
  | loTimerQueue: None

```

Snippet 10 shows an example of the simulator interface, where all relevant queues and registers are displayed for each processor.

4.2 Writing Transputer Programs in Assembly

Programs written for the Transputer can be wildly different from those written for popular modern assembly languages due to the evaluation stack and concurrency support. Here, we shall analyse the Transputer instruction set from the perspective of a potential assembly programmer.

4.2.1 Static Chains

It is often convenient to pass a pointer to the original workspace to allow procedures to access local variables in a different scope. In the following program, we have implemented a Fibonacci sequence calculator, using the following iterative algorithm:

Algorithm 4 Iterative Fibonacci

```
function MAIN
    a := 0
    b := 1
    n := 0

    function ITERFIB
        temp := b
        b := a + b
        a := temp

    while n ≠ 20 do
        ITERFIB()
        n := n + 1
```

The inner function ITERFIB needs access to a and b from the outer scope MAIN, thus a static link is required.

We may write the program in assembly like so:

Snippet 11 Fibonacci iteration using a procedure call

```
1  %istart 0x80000060
2  %wstart 0x80000200
3
4  .addr 0x80000060
5      ldc 0; stl 1     # a in w+1
6      ldc 1; stl 2     # b in w+2
7      ldc 0; stl 3     # n in w+3
8  loop:
9      ldlp 0
10     call iterfib
11     ldl 3; adc 1; stl 3       # increment n
12     ldl 3; eqc 20; cj loop    # break if n = 20
13     .break
14
15 iterfib:
16     ajw -1           # Leave one space for temp store
17     ldl 2; ldnl 1    # Load a
18     ldl 2; ldnl 2    # Load b
19     stl 0; ldl 0     # Save temp = b
20     add              # b' = a + b
21     ldl 2; stnl 2    # Store b'
22     ldl 0
23     ldl 2; stnl 1    # Store a' = temp in new position
24     ajw 1
25     ret
```

Static link tracing is done with `ldl 2`, followed by either `ldnl` (Lines 17, 18) or `stnl` (21, 23) for loading or saving.

4.2.2 Dynamic Procedure Calls

As explained in 2.1.4, calling procedures passed as arguments requires the use of the `gcall` instruction and a stub procedure. We can demonstrate this in the following program, where we calculate a Collatz Conjecture sequence, defined as such:

$$x_{n+1} = \begin{cases} \frac{x_n}{2} & \text{if } x_n \equiv 0 \pmod 2 \\ 3x_n + 1 & \text{if } x_n \equiv 1 \pmod 2 \end{cases}$$

It has been conjectured that any starting number x_0 eventually reaches 1 [12]. We have chosen an arbitrary starting number 39.

The pseudocode is as follows:

Algorithm 5 Collatz with Dynamic Procedure Calls

```
function Main
    x := 39
    n := 0

    function Inc
        x := 3x + 1

    function Dec
        x := x / 2

    function IterCollatz(Even, Odd)
        if x ≡ 0 (mod 2) then Even()
        else Odd()

    while x ≠ 1 do
        IterCollatz(Dec, Inc)
        n := n + 1
```

Both INC and DEC require access to x, and hence need a static link that point back to the main scope (Fig. 31):

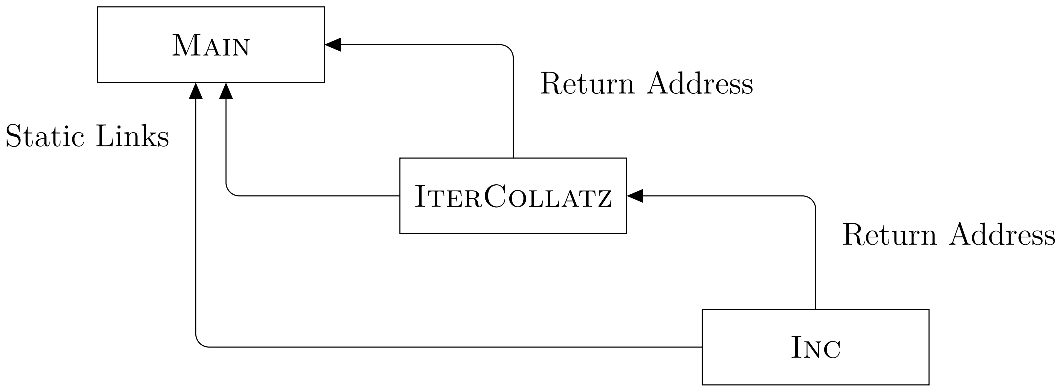


Figure 31: Workspace Structure when calling INC

Due to there only being 3 stack registers, we have chosen to pass a single static link as opposed to the convention of bundling a separate link to each passed procedure.

The main procedure, as well as the **inc** and **dec** procedures that modify x, are defined as such:

Snippet 12 Main procedure in Collatz program

```
%istart 0x80000060
%wstart 0x80000200

.addr 0x80000060
    ldc 39; stl 1           # x = 39 in w+1
    ldc 0; stl 2            # counter in w+2
loop:
    ldc inc                 # Odd proc: param 3
    ldc dec                 # Even proc: param 2
    ldlp 0                  # Static link: param 1
    call itercollatz
    ldl 2; adc 1; stl 2
    ldl 1; eqc 1; cj loop
    .break

inc:
    ldl 1; ldnl 1           # Follow static chain to previous term
    ajw -1; stl 0;          # Allocate one word
    ldl 0; ldc 1; shl       # x' = x << 2
    ldl 0; add              #      + x
    adc 1                   #      + 1
    ajw 1                   # Deallocate
    ldl 1; stnl 1           # Follow static chain to store new term
    ret

dec:
    ldl 1; ldnl 1
    ldc 1; shr              # x' = x >> 2
    ldl 1; stnl 1
    ret
```

When defining **itercollatz**, we will need an additional **stub** procedure:

Snippet 13 Iteration procedure in Collatz program (Continued from previous snippet)

```
32 stub:
33     ldl 2                    # Load param 2
34     gcall                    # Call passed function
35
36 itercollatz:
37     ldl 1; ldnl 1            # Load previous term
38     ldc 1; and               # Mask LSB
39     eqc 1; cj even
40 odd:
41     ldl 3                    # Odd proc: param 3
42     ldl 1                    # Static chain: param 0
43     call stub
44     ret
45 even:
46     ldl 2                    # Even proc: param 2
47     ldl 1                    # Static chain: param 0
48     call stub
49     ret
```

An indirect call to one of the passed procedures in **itercollatz** involves calling **stub** (32–34) first, which swaps the procedure reference into *Iptr* using **gcall** (34).

4.3 Concurrent Programs

We can facilitate communication between processes using the following instructions (Table 9), which send and receive messages using channels:

Instruction	`Areg`	`Breg`	`Creg`
`in`	Message Size (bytes)	Channel Address	Write to
`out`	Message Size (bytes)	Channel Address	Message Address
`outbyte`	Message Address	Channel Address	-
`outword`	Message Address	Channel Address	-

Table 9: Channel instructions

`outbyte` and `outword` have fixed message sizes of 1 byte and 1 word respectively.

External link ports also have fixed positions in the address space (Table 10):

Port	Outgoing (Sending)	Incoming (Receiving)
0	`0x80000000`	`0x80000010`
1	`0x80000004`	`0x80000014`
2	`0x80000008`	`0x80000018`
3	`0x8000000c`	`0x8000001c`

Table 10: Channel addresses

These addresses can be quickly loaded onto the stack using the `mint` (Minimum Integer) instruction, which pushes `0x80000000`.

4.3.1 Guarded Alternatives

We can reimplement the Collatz Program earlier using the following process defined in CSP:

$$C(x_0) = P(x_0) \underset{\{|\text{num},\text{odd},\text{even}|\}}{\|} Q$$

$$P(x) = \text{num}!x \rightarrow \left(\text{odd} \rightarrow P(3x+1) \,\square\, \text{even} \rightarrow P\left(\frac{x}{2}\right)\right)$$

$$Q = \text{num}?x \rightarrow ((x \equiv 0 \pmod 2) \,\&\, \text{even} \rightarrow Q) \,\square\, (x \equiv 1 \pmod 2) \,\&\, \text{odd} \rightarrow Q))$$

The program begins by first initialising the three required channels, before forking[2] to form two separate processes, P and Q.

Snippet 14 Initialising channels for Guarded Alts Collatz

```
1  %istart 0x80000060
2  %wstart 0x80000300
3  
4  .addr 0x80000060
5      mint; ldc num_chan;
6      stl 2; ldl 2; stnl 0          # Initialize num_chan
7      mint; ldc odd_chan;
8      stl 3; ldl 3; stnl 0          # Initialize odd_chan
9      mint; ldc even_chan;
10     stl 4; ldl 4; stnl 0          # Initialize even_chan
11     ldc 2; stl 1                  # Store process count on w+4
12     ldc end - p_ref; ldpi         # Relative jump from ref to end
13 p_ref:
14     stl 0                         # Store end address
15     ldc proc_q - q_ref
16     ldlp q_ws - p_ws
17     startp                        # Start process Q
18 q_ref:
19     j proc_p
```

[2]Forking with **startp** requires us to store a program end address (14) and total process count (11) locally.

As introduced in 2.2.6, the alternatives in P are enabled (28-29) and disabled (31-32) in order, with `altend` jumping to the chosen branch (`p_odd` or `p_even`).

Snippet 15 Process P (Continued from previous snippet)

```
22 proc_p:
23     ldc 39; stl 5                # Term on w+5
24     ldc 0; stl 6                 # Term index on w+6
25 p_loop:
26     ldl 2; ldl 5; outword        # num_chan!term
27     alt
28     ldl 3; ldc 1; enbc           # Enable odd_chan alt
29     ldl 4; ldc 1; enbc           # Enable even_chan alt
30     altwt
31     ldl 3; ldc 1; ldc 0; disc                # Disable odd_chan alt
32     ldl 4; ldc 1; ldc p_even - p_odd; disc   # Disable even_chan alt
33     altend
34 p_odd:
35     ldlp 7; ldl 3; ldc 4; in     # odd_chan ? x
36     ldl 5; ldc 3; mul            # x * 3
37     ldc 1; add                   #   + 1
38     stl 5                        # save new x
39     j p_cond
40 p_even:
41     ldlp 7; ldl 4; ldc 4; in     # even_chan ? x
42     ldl 5; ldc 1; shr            # x >> 1
43     stl 5                        # save new x
44     j p_cond
45 p_cond:
46     ldl 6; adc 1; stl 6          # Increment index
47     ldl 5; eqc 1; cj p_loop      # Check if term is 1
48 end:
49     .break
50     .byte 0
```

Meanwhile, Q does the odd/even check and branches with a conditional jump (64):

Snippet 16 Process Q (Continued from previous snippet)

```
53 proc_q:
54     ldc num_chan; stl 2          # num_chan on w+1
55     ldc odd_chan; stl 3          # odd_chan on w+2
56     ldc even_chan; stl 4         # even_chan on w+3
57 q_loop:
58     ldlp 1                       # Receiving address w+1
59     ldl 2                        # Target channel
60     ldc 4                        # Message size (4 bytes)
61     in
62     ldl 1                        # Load message
63     ldc 1; and                   # Mask LSB
64     eqc 1; cj q_even             # Check even/odd
65 q_odd:
66     ldl 3; ldc 1; outword        # odd_chan!1
67     j q_loop
68 q_even:
69     ldl 4; ldc 1; outword        # even_chan!1
70     j q_loop
```

Lastly, we allocate space for all the processes and channels:

Snippet 17 Workspace Allocation (Continued from previous snippet)

```
.addr 0x80000300
p_ws:
    .zero 128
q_ws:
    .zero 128
num_chan:
    .word 0
odd_chan:
    .word 0
even_chan:
    .word 0
```

4.3.2 Bag of Tasks

To demonstrate guarded alternatives, we turn to the bag of tasks idiom for dividing tasks [13]. The following program multiply two matrices using 5 processors (Fig. 32):

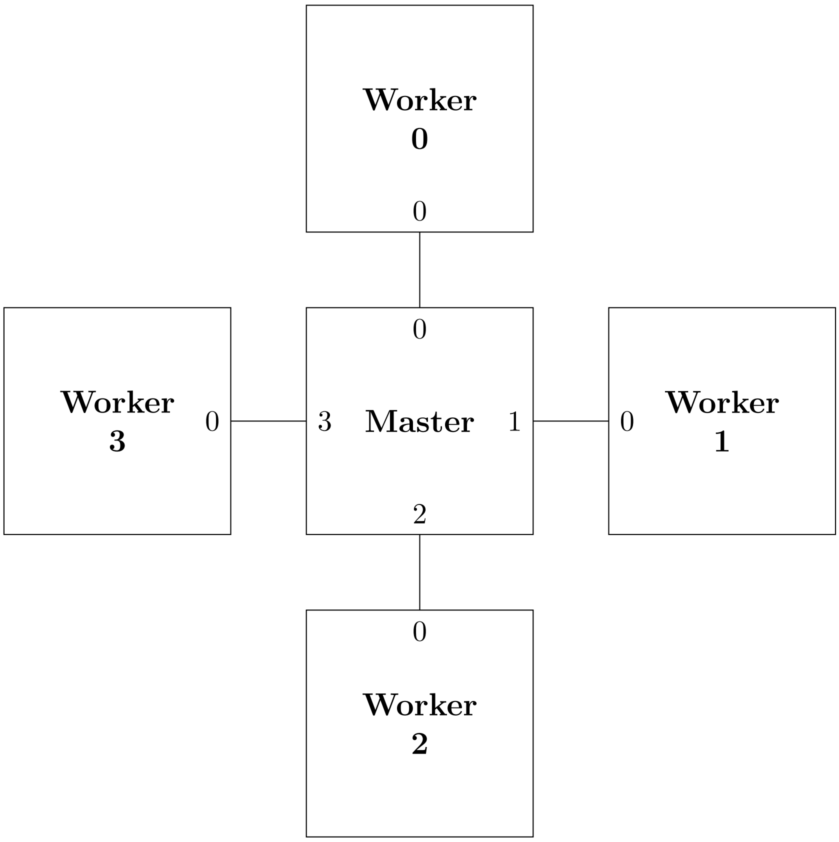


Figure 32: Port connections

Each task involves calculating the dot product of two vectors, resulting in a single value in the result matrix.

We first analyse the program flow of each worker:

Snippet 18 Matrix Multiplication Worker Program

```
1  %istart 0x80000060
2  %wstart 0x80000200
3
4  .addr 0x80000060
5  main:
6      mint; stl 1                      # Link 0 Out at w+1
7      mint; adc 0x10; stl 2            # Link 0 In at w+2
8      ldc data_left; stl 3             # Data storage 1
9      ldc data_top; stl 4              # Data storage 2
10     ldlp 5; ldl 2; ldc 4; in         # Receive task size in w+5
11 loop:
12     ldl 3; ldl 2; ldl 5; in          # Receive first data batch
13     ldl 4; ldl 2; ldl 5; in          # Receive second data batch
14     ldc 0; stl 6                     # Relative pointer at w+6
15     ldc 0; stl 7                     # Sum at w+7
16 mul_loop:
17     ldl 3; ldl 6; add; ldnl 0        # Load multiplicand
18     ldl 4; ldl 6; add; ldnl 0        # Load multiplier
19     mul; ldl 7; add; stl 7           # Multiply and add to sum
20     ldl 6; ldnlp 1; stl 6            # Increment pointer
21     ldl 5; ldl 6; diff               # total - evaluated
22     cj end
23     j mul_loop
24 end:
25     ldl 1; ldlp 7; outword            # Send sum back
26     j loop
27
28
29 .addr 0x80000300
30 data_left:
31     .zero 256
32 data_top:
33     .zero 256
```

Workers are initialised with a vector size (10), before they repeatedly receive the pairs of vectors to multiply (11-23). Results are sent back (25) before the worker waits for the next task.

It helps to keep track of workspace usage like so (Table 11):

Location	Usage
`w+5`	Vector size (Task size)
`w+4`	Pointer to Multiplier Vector
`w+3`	Pointer to Multiplicand Vector
`w+2`	Pointer to External Link 0 (In)
`w+1`	Pointer to External Link 0 (Out)
`w+0`	Unused

Table 11: Worker Program Workspace Usage

Workspace usage on the master processor is listed in table 12 below. Note that we refer to each of the 4 ports by *port offsets*, which range from `0x00` for Port 0 to `0x0c` for Port 3.

Location	Usage
`w+17`	Worker 3 current task tracker (identified by result address)
`w+16`	Worker 2 current task tracker
`w+15`	Worker 1 current task tracker
`w+14`	Worker 0 current task tracker
`w+13`	Most recently received task
`w+12`	Result matrix end pointer
`w+11`	Result matrix current pointer
`w+10`	Matrix 2 end pointer
`w+9`	Matrix 2 current pointer
`w+8`	Matrix 1 end pointer
`w+7`	Matrix 1 current pointer
`w+6`	Result matrix start pointer
`w+5`	Matrix 2 start pointer
`w+4`	Matrix 1 start pointer
`w+3`	Matrix size (in bytes)
`w+2`	Row size (in bytes)
`w+1`	Row length (number of values)

Table 12: Worker Program Workspace Usage

All necessary pointers are prepared first:

Snippet 19 Matrix Multiplication Master Program

```
%istart 0x80000060
%wstart 0x80000400

.addr 0x80000060
main:
    ldc 32; stl 1                # Dimensions stored in w+1
    ldc 0; ldnlp 4; stl 2        # Dimensional jump stored in w+2
    ldl 2; ldl 1; mul; stl 3     # Dimension limit stored in w+3
    ldc data_left; stl 4         # Data storage 1 (Stored row-wise)
    ldc data_top; stl 5          # Data storage 2 (Stored column-wise)
    ldc data_result; stl 6       # Result storage (Stored row-wise)
    ldl 4; stl 7                 # Storage 1 data pointer at w+7
    ldl 4; ldl 3; add; stl 8     # Storage 1 end pointer at w+8
    ldl 5; stl 9                 # Storage 2 data pointer at w+9
    ldl 5; ldl 3; add; stl 10    # Storage 2 end pointer at w+10
    ldl 6; stl 11                # Result storage pointer at w+11
    ldl 6; ldl 3; add; stl 12    # Result end pointer at w+12
    ldc 0; stl 13                # Most recently received result addr at w+13

    # Initialize matrices
    ldlp 0; call gen_matrix

    # Initialize workers with dimensions
    mint; ldlp 2; outword
    mint; adc 0x04; ldlp 2; outword
    mint; adc 0x08; ldlp 2; outword
    mint; adc 0x0c; ldlp 2; outword

    # Assign work to workers
    ldc 0x00; ldlp 0; call send_worker
    ldc 0x04; ldlp 0; call send_worker
    ldc 0x08; ldlp 0; call send_worker
    ldc 0x0c; ldlp 0; call send_worker
```

Note that we have chosen to store the multiplicand in row-major order and the multiplier in column-major order to simplify the code.

To provide the program with two matrices to multiply, we have written the procedure **gen_matrix**, which implements the following algorithm:

Algorithm 6 Matrix Generation

```
function GenMatrix
    mat1_ptr := Matrix 1 start pointer
    mat2_ptr := Matrix 2 start pointer
    row_first := 1
    data := 1

    for row = 32 ... 1 do
        for col = 32 ... 1 do
            *mat1_ptr := data
            *mat2_ptr := data
            data := data + 1
            mat1_ptr := mat1_ptr + 1
            mat2_ptr := mat2_ptr + 1

        row_first := row_first + 1
        data := row_first
```

This produces the same constant matrix for both sides of the multiplication:

$$\begin{bmatrix} 1 & 2 & \cdots & 32 \\ 2 & 3 & \cdots & 33 \\ \vdots & \vdots & \ddots & \vdots \\ 32 & 33 & \cdots & 63 \end{bmatrix}$$

Implemented in Transputer assembly:

Snippet 20 Matrix Generation

```
gen_matrix:
    ajw -6                                  # Allocate 6 words
    ldl 7; ldnl 7; stl 0                    # Matrix 1 pointer at w+0
    ldl 7; ldnl 9; stl 1                    # Matrix 2 pointer at w+1
    ldl 7; ldnl 1; stl 2                    # Row at w+2
    ldl 2; stl 3                            # Col at w+3
    ldc 1; stl 4                            # Row first num at w+4
    ldl 4; stl 5                            # Data at w+5
gen_col_loop:
    ldl 5; ldl 0; stnl 0                    # Write data to matrix 1
    ldl 5; ldl 1; stnl 0                    # Write data to matrix 2

    ldl 5; adc 1; stl 5                     # Increment data
    ldl 0; ldnlp 1; stl 0                   # Increment matrix 1 pointer
    ldl 1; ldnlp 1; stl 1                   # Increment matrix 2 pointer
    ldl 3; adc -1; stl 3                    # Decrement col

    ldl 3; cj gen_row_loop                  # If col > 0, back to gen_col_loop
    j gen_col_loop
gen_row_loop:
    ldl 7; ldnl 1; stl 3                    # Reset col
    ldl 2; adc -1; stl 2                    # Decrement row
    ldl 4; adc 1; stl 4                     # Inc row first num
    ldl 4; stl 5                            # Set data to row first num

    ldl 2; cj gen_col_loop                  # If row > 0 then jump back
    ajw 6
    ret
```

The `send_worker` procedure is used to distribute a task to a worker:

Snippet 21 Worker Task Distribution

```
112 send_worker:
113     ldl 1; ldnl 7                          # Get data address
114     mint; ldl 2; add                       # Get channel
115     ldl 1; ldnl 2                          # Get size (in bytes)
116     out                                    # Send multiplicants
117
118     ldl 1; ldnl 9
119     mint; ldl 2; add
120     ldl 1; ldnl 2
121     out                                    # Send multipliers
122
123     ldl 1; ldnl 7
124     ldl 1; ldnl 2; add                     # Increment row pointer
125     ldl 1; stnl 7
126
127     ldl 1; ldnl 11                         # Get task result pointer
128     ldl 1; ldnlp 12; ldl 2; add; stnl 0    # Assign task result pointer to worker
129     ldl 1; ldnl 11
130     ldl 1; ldnl 2; add                     # Increment result pointer
131     ldl 1; stnl 11
132
133     # If row pointer has reached the end, reset to start
134     ldl 1; ldnl 8
135     ldl 1; ldnl 7; diff
136     cj inc_col
137     ret
138
139 inc_col:
140     ldl 1; ldnl 4; ldl 1; stnl 7
141     ldl 1; ldnl 9                          # Reset col index
142     ldl 1; ldnl 2; add
143     ldl 1; stnl 9                          # Increase row
144     ret
```

After task assignment, it updates the current pointers to the next available task (113–121), resetting the column index where necessary (133–144). It then updates the task tracker for the worker in question (127–131).

After distributing the first set of tasks, the master program then enters its main loop, using a guarded alternative to wait for the next available result.

Snippet 22 Master Program Main Loop

```
35 loop:
36     alt
37     mint; adc 0x10; ldc 1; enbc
38     mint; adc 0x14; ldc 1; enbc
39     mint; adc 0x18; ldc 1; enbc
40     mint; adc 0x1c; ldc 1; enbc
41     altwt
42     mint; adc 0x10; ldc 1; ldc 0; disc
43     mint; adc 0x14; ldc 1; ldc 0; disc
44     mint; adc 0x18; ldc 1; ldc 0; disc
45     mint; adc 0x1c; ldc 1; ldc 0; disc
46     altend
47 
48 worker_a:
49     ldc 0x00; j distribute
50 worker_b:
51     ldc 0x04; j distribute
52 worker_c:
53     ldc 0x08; j distribute
54 worker_d:
55     ldc 0x0c
56 distribute:
57     ldlp 0
58     call receive_result          # p0: static link, p1: worker offset
59     ldl 12; ldl 11; gt; cj skip_send
60     ldlp 0
61     call send_worker             # Send task if there are still tasks
62 skip_send:
63     ldl 13; ldnlp 1; ldl 12; diff
64     cj end                       # If all tasks are done, end.
65     j loop                       # Else go back
66 
67 end:
68     .break
69     .byte 0
```

Each branch pushes the relevant port offset onto the stack (48-55) before the corresponding result is retrieved (58). We check if there are new tasks by checking if the current pointer has reached the end (59), before sending a new task to the same worker (61).

This setup fails to guarantee fairness; if worker 0 completes its tasks too quickly, it will also receive the next task. Fairness would require dynamically changing the enabling/disabling order.

After receiving each result, we use the current task trackers to figure out where to write the result to:

Snippet 23 Receiving Result Products

```
101 receive_result:
102     ajw -1
103     ldl 2; ldnlp 14; ldl 3; add; ldnl 0      # Get data offset
104     ldl 2; stnl 13; ldl 2; ldnl 13           # Save data offset
105     mint; adc 0x10; ldl 3; add               # Get in channel
106     ldc 4                                    # Word size (in bytes)
107     in
108     ldl 3                                    # Save branch to stack
109     ajw 1
110     ret
```

We also have to be mindful of preserving the port offset (108), to pass it to `send_worker` later if necessary.

Lastly, space is allocated for 3 matrices:

Snippet 24 Allocating space for matrices

```
.addr 0x80000500
    data_left:
        .zero 2048
    data_top:
        .zero 2048
    data_result:
        .zero 2048
```

4.4 Evaluation

Having explored these programs, we are now in a better position to judge its many design decisions from a modern perspective. With hardware support for concurrency being a major focus of the processor, it had a lot of potential, which it had certainly fulfilled to some extent with its accomplishments. We have seen this through our bag of tasks example, but it can easily be expanded with a suitable compiler and code written for more interconnected Transputers.

Nevertheless, its use of single-byte instructions and an evaluation stack leaves much to be desired. Instructions dealing with large operands easily increase program size, and their inability to access specific registers severely limits their conciseness. Combined with a stack size of merely 3, many of the programs we have written are choke full of memory addresses, serving as a major bottleneck for any program. This is especially true when we compare the T414 to the architectures we have today, where values such as loop counters can easily be stored into registers.

5 Conclusion

This report provides a concise summary of the unique features supported by the T414, including its stack evaluation registers, procedure calling mechanisms, as well as concurrent processes facilitated by process queues, communication channels, and alternative guards.

A new syntax for Transputer assembly has been decided, in which programmers can write, assemble, and test programs using the tools provided in TAMS. Through its development, we have been provided with an opportunity to study the implications of the Transputer instruction set design, including label address calculation and prefix expansion. The development of the simulator also revealed more intricacies such as the mechanisms surrounding process management, multiprocessor stepping, and external link data transfers.

Finally, by testing a variety of programs on TAMS, we have gained further insight into the style of programs written in Transputer assembly, including the use of static chains, planned workspace allocations, and the coordination of concurrent programs through the use of channels. These programs have also exposed the downsides of the Transputer architecture, such as its stack machine design and excessive memory accesses.

5.1 Reflection

During the initial research process, finding relevant resources for the T414 turned out to be far harder than I had expected, due to how fragmented and convoluted the official documentation was, and how inaccessible they were compared to many of the online resources today. This was one of the main motivating factors for me to write the new summaries that constituted chapter 2 of this report - behind every diagram was hours of research time poured into parsing the obtuse texts in publications from INMOS.

On that note, I relate very much with John Roberts, the writer of "Transputer Assembler Language Programming", who noted back in 1992 that the frustration he had after "struggling with nonstandard and sometimes cryptic documentation from Inmos" had driven him to write an entire book [11]. Many key details essential to implementing a simulator have simply been left out, and the summaries I wrote proved to be extremely useful when I implemented TAMS.

Choosing to write TAMS in a language I'm comfortable with (C++) allowed me to focus on the implementation rather than the quirks of the language. Implementing the entire instruction set proved to be tedious and time-consuming, especially with complex instructions that Inmos merely described the behaviours of in vague prose.

However, the largest time sink turned out to be ensuring that multiprocessor simulations worked independent of declaration order. This involved making sure that all possible evaluation sequences in every possible channel configuration configuration was thoroughly tested. Since the code keeps an ordered array of processors and links which it iterates on each cycle, writing deterministic code that demonstrates the same behaviour regardless of the array order meant that extra care had to be put into program flows to allow interprocessor interactions to work independently of the order in which they are processed.

Writing example programs for TAMS turned out to be a highly creative process because of how different it was from what I was used to. Without random access to all the registers, these programs required a lot more planning, especially with workspace usage and alternative guard branches. This was necessary to prevent the number of memory accesses from being further inflated. The slowness of memory access is a major focal point for a lot of modern optimisation, but the apparent indifference to it on the Transputer and the sheer number of registers it had dedicated to managing concurrent processes instead really spoke to the lengths its designers have gone to push their new vision of parallel processing.

5.2 Future Work

By referring to the implementation details in TAMS, it would be possible to attempt to recreate the processor in hardware using FPGAs. However, it would be far more interesting to explore the unrealised potential of the Transputer by designing an architecture and instruction set inspired by the concurrency support of the T414, without the limitations of a stack machine.

References

[1] INMOS Limited, *IMS T414 engineering data*, 2nd ed. INMOS Limited, 1989, pp. 333–398.

[2] The Go Project, “Frequently asked questions (faq).” [Online]. Available: https://go.dev/doc/faq

[3] G. Crate, “Transputer emulator.” [Online]. Available: https://sites.google.com/site/transputeremulator/Home

[4] A. Pahi, “T4.” [Online]. Available: https://github.com/pahihu/t4

[5] G. Crate, “Jserver emulator.” [Online]. Available: https://sites.google.com/site/transputeremulator/Home/jserver

[6] J. C. Highfield, “Inmos t414 transputer emulator.” [Online]. Available: https://www.macintoshrepository.org/2118-inmos-t414-transputer-emulator

[7] INMOS Limited, *Transputer instruction set: a compiler writer’s guide.* New York; London: Prentice-Hall, 1988.

[8] D. A. P. Mitchell, *Inside the Transputer*, ser. Computer Science Texts. Oxford: Blackwell Scientific, 1990.

[9] C. A. R. Hoare, “Communicating sequential processes,” *Commun. ACM*, vol. 21, no. 8, aug 1978. [Online]. Available: https://doi.org/10.1145/359576.359585

[10] A. W. Roscoe, *Understanding Concurrent Systems*, ser. Texts in Computer Science. New York; London: Springer, 2010.

[11] J. Roberts, *Transputer Assembly Language Programming.* Van Nostrand Reinhold Computer, 1992.

[12] Ş. Andrei and C. Masalagiu, "About the collatz conjecture," *Acta Informatica*, vol. 35, no. 2, pp. 167–179, 1998.

[13] G. R. Andrews, *Foundations of Multithreaded, Parallel, and Distributed Programming.* Reading, Massachusetts; Harlow: Harlow, 2000.