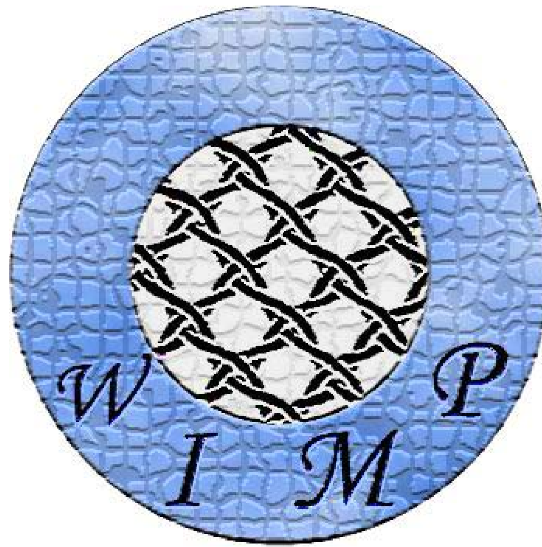


Wisconsin's Interleaved Multithreaded Processor

Software Manual

Bryan Berns
Jacob Petranak
Jordan Wenner
Parikshit Narkhede
Suman Mamidi

Table of Contents

Assembler 3
WIMP-IDE (WIMP Development Environment) 9
WIMPTERM (WIMP Terminal Client) 10
Protocol For Image Transmission: Receiving From PC 12
Protocol For Image Transmission: Sending From PC 12
WIMP Software Architecture 13
WIMPOS Kernel 14
I/O With Multiple Threads 14
Printing to SPAT 15
Reading from keyboard 16
Passing Parameters To Functions 16
Getting a file from the PC 17
Sending a binary file to PC 18
Servicing shell commands 19
Updating The VGA 19
Image Processing Applications 20
Thresholding 20
Bit Plane Slicing 22
Edge Detection 23
Time Redundant Fault Tolerant Applications 26

Assembler

The assembler for WIMP is very versatile. It supports symbols and multiple address code mappings. To anticipate design changes in the instruction set, it gains its instruction knowledge from the instruction documentation itself. The assembler is written in C++ and is compiled into one executable for simplicity.

Part 1: Numerical Addressing & Comments

The WIMPy assembler includes support for strings, hexadecimal integers and decimal integers. Any statement requiring an integer input will parse the integer for a '0x' precedent. If the number is preceded with a '0x', the following decimal numbers will be treated as base 16. If it is not present, base 10 will be assumed. For example,

Using hexadecimal: 0xF
Equivalent decimal: 15

Strings can only be used in the .data section, as elaborated below. Strings are identified by being surrounded with quotation marks. They can contain three useful escape characters: \n, \t, \0. The \n will evaluate to the byte value for a 'newline'. The \t will evaluate to the byte value for a tab. The \0 will evaluate to a null byte value (0), which may be useful for a produced to know when the string has ended.

Comments are indicated by the pound symbol: # There is no multi-line comment indicator, each line must be preceded with #.

Part 2: Coding Sections

The WIMP assembler includes three section directives: code, section, include. The 'include' statement is actually a directive, but it must be used in conjunction with a section.

`.data ADDR`

The .data section indicates any following non-comment lines will be interpreted as data variables, which has a special syntax. The ADDR section specifies where the data section begins. A .data section is NOT needed for each variable, as multiple variables' addresses are calculated by ADDR + sizeof(previous_varabiles). For example:

```
.data 0x4000    # This means the first variable
                # will appear at memory address 0x4000
string_one:0x40 # This variable begins at 0x4000
integer_one:2   # This variable appears at 0x4042
integer_two:2   # This variable appears at 0x4044
```

Again notice the interchangeability of hexadecimal and decimal integers. One can also indicate an initial value for a variable by following the byte size parameter with '='. For example:

```
.data 0x4000
string_one:0x40="Name: Tom\n"
integer_one:2=16
integer_two:2=0x10
```

Giving variable initial variables will allow the assembler to output a memory file for the XESS board with these values. Please note if more than two bytes is specified is 'initialized' to an integer values, the assembler will put the value into the first two bytes.

Aside from strings, the assembler can also load binary files when prefixed with a '<' character. For instance, the following code load the binary file 'image16x16.bin' into the address 0x5000:

```
.data 0x5000
string_one:256=<images/image16x16.bin
```

If the file is less than the data allocation the values will be filled with don't cares or 0xFF, depending on the file output. This is also the case for over-allocated strings.

.code ADDR

Similar to data, the .code statement takes an address which indicates the first instruction it encounters is at that addresses. Following memory location of following addresses is calculates as ADDR + 2*number_of_previous_instructions. There are three main types of instructions which gain more specific meaning via the instruction file, which is a tab delimited output of WIMP's ISA Excel file. The details of the parsing need not be discussed, but it is useful to know the assembler will check to see if a parameter begins with a '$' if the instruction require a register. For example:

```
.code 0x2000       # This means the first instruction
                   # will appear at memory address 0x2000
add.16 $2,$2,$3    # Begins at 0x2000
li.hi 0xF,$2       # Begins at 0x2002
li.lo 30,$2        # Begins at 0x2004
```

It is also legal to replace the comma between parameters with a space. Thusly 'add.16 $2 $2 $2' would also be legal syntax.

To make a symbol (or label) in the .code section, the syntax is similar to .data but without the size and initial values:

```
.code 0x2000     # This means the first instruction
                 # will appear at memory address 0x2000
add.16 $2,$2,$3 # Begins at 0x2000
```

```
loop:           # The loop symbol will resolve to 0x2002
li.hi 0xF,$2    # Begins at 0x2002
li.lo 30,$2     # Begins at 0x2003
```

.include FILE

The include statement does exactly that of any complier: it simply inserts the content of ‘FILE’ into the source file exactly where the line includes. To demonstrate this functionality, the following is an example should be assembler into the exact same binary as the two previously given code/data sections:

```
.include /data_file
.include /code_file
```

```
Contents of data_file:

.data 0x4000
string_one:0x40="Name: Tom\n"
integer_one:2=16
integer_two:2=0x10
```

```
Contents of code_file:

.code 0x2000
add.16 $2,$2,$3
li.hi 0xF,$2
li.lo 30,$2
```

Also note the include directive will inherit the .code section or .data section if one is not explicitly given in it. For example, the following is equitable to the above.

```
.data 0x4000
.include /data_file
.code 0x2000
.include /code_file
```

```
Contents of data_file:

string_one:0x40="Name: Tom\n"
integer_one:2=16
integer_two:2=0x10
```

```
Contents of code_file:

add.16 $2,$2,$3
li.hi 0xF,$2
li.lo 30,$2
```

This functionality makes coding easier if similar data (or code) need to be used in both system and user memory.

.thread IMM

The .thread statement has no effect unless an assembler macro (explained later) is being used. It accepts a valid thread number, ideally specifying in which thread the code is being processed. The .thread statement is processed linearly while evaluating code so if an .include statement specifies a file which contains a .thread statement, it will overwrite the value specified by .thread in the calling file.

Part 3: Psuedo-Instructions

Psuedo Instruction	Assembly Equivalent	Description
`la symbol, $register`	`li.lo lobyte(addressof(symbol)),$register` `li.hi hibyte(addressof(symbol)),$register`	The load address pseudo work is for loading non-immediate jumps and memory addresses into code. The first parameter is the symbol as explained in the .data or .code sections and the register is where to store the address.

`lwr symbol, $register`	`li.lo lobyte(addressof(symbol)),$register`
	`li.hi hibyte(addressof(symbol)),$register`
	`lw $register,$register`
		The load word register pseudo instruction is used for loading immediate values stored at the symbol into the code. The first parameter is the symbol as explained in the .data or .code sections and the register is where to store the immediate value.

`lwr symbol, $register`	`li.lo lobyte(addressof(symbol)),$register`
	`li.hi hibyte(addressof(symbol)),$register`
	`lw $register,$register`
	`lw $register,$register`
		The load word from pointer pseudo instruction is used for loading the at a memory address which is contained in memory. It is similar another level of abstraction from the load word from register instruction.

`li IMM, $register`	`li.lo lobyte(IMM),$register` `li.hi hibyte(IMM),$register`	The load immediate pseudo instruction simply loads a two-byte immediate into a register. It simplifies cases in which both halves of the register are being used (word-mode).
`jmp symbol`	`jmpi addressof(symbol) - currentaddress(),COND`	The jump instructions will assemble into a jump immediate instruction using the offset value to the desired symbol. Offset cannot be more than 256 or the assembler will return an error. It will only jump the COND bit is set in the flag register.
`jal symbol`	`jali addressof(symbol) - currentaddress(),COND`	The jump and link instructions will assemble into a jump and link immediate instruction using the offset value to the desired symbol. Offset cannot be more than 256 or the assembler will return an error. It will only jump the COND bit is set in the flag register.

* The lobyte(),hibyte(),addressof() and currentaddress() are assembler functions which return immediate values at the time of assembly depending on the parameters passed.

Part 4: Data Output and Syntax

All parameters passed to the assembler need to be quoted if they contain spaces. This is standard for all command line programs in general.

WIMPASM [ISA] [ASM] [MEMLOW] [MEMHIGH] [XES] [CONSOLE] [IMAGE]

ISA: (Input)

The tab-delimited output of WIMP's ISA in excel.

ASM: (Input)

The assembly file. You can use the .include directive in the ASM file if you wish to assemble multiple files in one.

MEMLOW: (Output)

A file containing a memory footprint for all code and initialized data for the assembled ASM file. It will only operate on data in ranges 0x0000 through 0x7FFFF. Note this file contains lines of 4 bytes of data so all unaligned data is appended with 0xFF values. This file type, aside from being very readable, is used for ModelSIM and/or other verilog simulators.

MEMHIGH: (Output)

Same as MEMLOW but for address 0x8000 through 0xFFFF

XES: (Output)

A memory footprint containing all code and initialized data for the assembled ASM file. Note this file contains lines of 16 bytes so any unfilled bytes are filled with 0xFF values. Any memory values which address memory above 0x8000 are re-positioned to start at 0x100000 (still relative to the offset at 0x8000). For the WIMP processor, all memory above 0x8000 is actually on the second chip and the XESS memory loading utility requires this memory translation to appropriate this.

CONSOLE: (Output)

A file containing an proprietary format for simple debugging of the assembler.

IMAGE: (Input)

A 128 KB initial image in RAW format to be loaded into the video ram.

Part 5: Assembler Macros

Assembler macros handle what an operating system might refer to as 'system calls'. Macros depend (and assume) specified symbol names and memory values are being included during the assembly. Most macros are used in conjunction with the .thread statement. Some macros use other macros and the following table describes each of these, in order of precedence.

Macro Instruction	**Description**
PASSUP V0,V1,V2…,$T1,T2	Pushes the specified parameters onto a thread's function stack. The values passed can be immediate values, a symbol, or a register.
PASSDOWN $R0,$R1,$R2…,$T1,$T2	Pops 'return' values off of a threads function stack.
PRINT S,$T1,$T2	Prints the null-delimited string located at SYMBOL.
PRINTR $R,$T1,$T2	Prints the 2 byte value of the register in hexadecimal format.
MEMCMP S1,S2,V0,$T1,$T2	If V0 is zero, then this function is equilvant to the C-code of strcmp(SYMBOL1,SYMBOL2). Otherwise, it is equilvant to memcmp(SYMBOL1,SYMBOL2,V0). If the comparison is true, 0 is retruned in $T1.
MAC $R1,$R2,$R3,V0,$T1,$T2	This function does implements a signed version of the processor's multiply and accumlate instruction. If V0 is zero, the function peformes a mac.lo, otherwise a mac.hi. The value is returned in $R3.
GETCHAR $R1,$T1,$T2	This function is a blocking call which waits for a character to be typed and stores it in $R1 before returning.
SEND_TO_VGA S,V0,$T1,$T2	This functions copies 1024 bytes located at the address of S to the section specified by V0.

Note: In the above table, the following prefixes are used:

R: general register
T: temporary register
V: label, symbol, or immediate
S: symbol

WIMP-IDE (WIMP Development Environment)

WIMP-IDE is based on GNU-Emacs Text Editor. It is developed in Emacs Lisp as an extension.

Some of the key features supported by WIMP-IDE are:

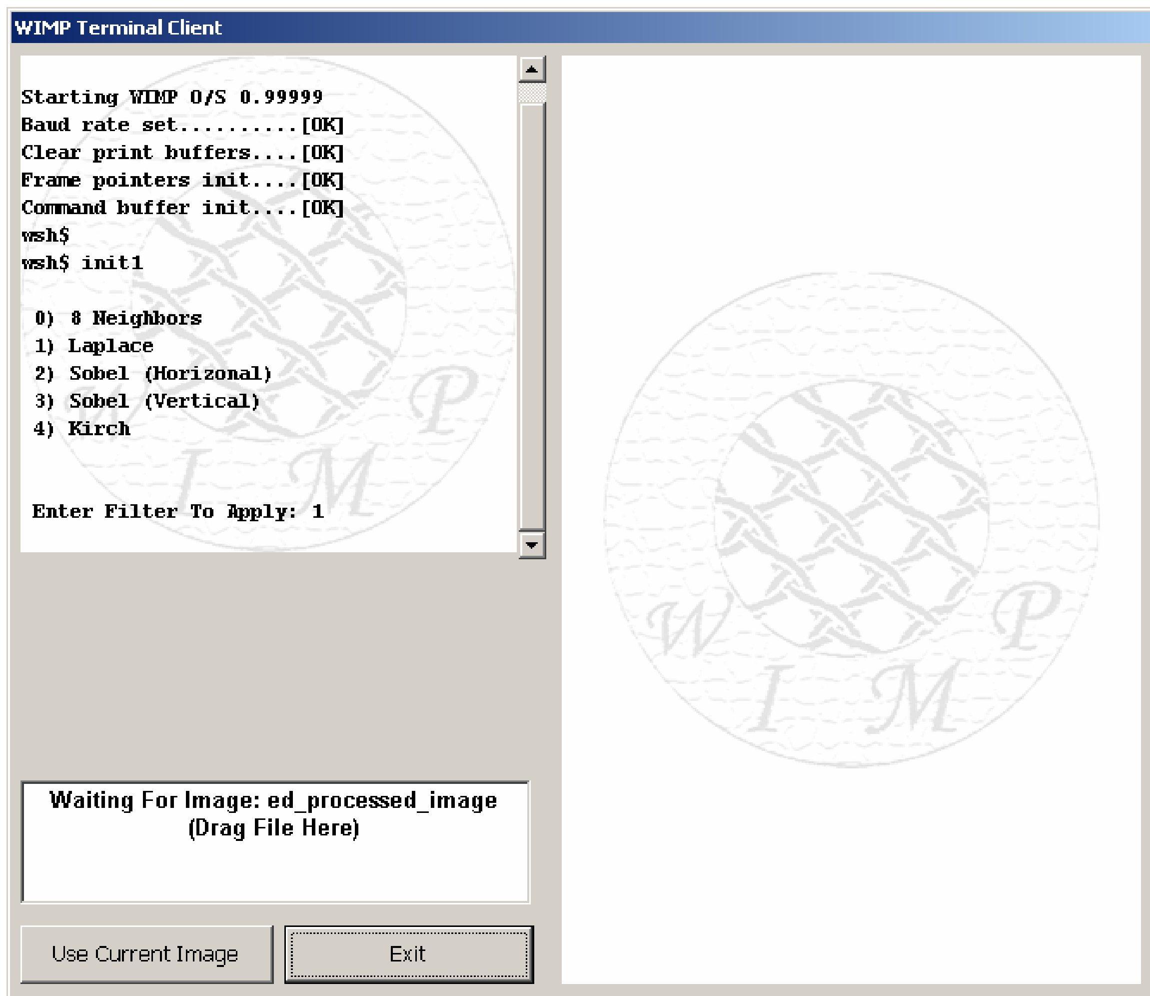
1. Syntax highlighting for instructions and pseudo-instructions
2. Auto-indentation of the source code.
3. Assembling the code using WIMPASM
4. Ability to communicate with XESS tools to perform
    a. Upload bit file to FPGA (with current program binary)
    b. Clear the LPT port
5. Can start HyperTerminal with any desired pre-saved configuration

```
emacs@CMPELABS-42
File  Edit  Options  Buffers  Tools  WIMP ASM  Help

WIMPy Assemble          (C-c C-a)
XSTOOLs                 >   Re-Configure FPGA   (C-c C-r)
                            Reset LPT
Start HyperTerminal     (C-c C-t)
Submit bug report
Version

.include include\init_i
.code 0x3000
TEST_SYSTEM_CODE:

.code 0x4000
USER_CODE:
.data 0x4000
THREAD_1:
THREAD_2:
THREAD_3:
.thread 1
li 0x01FE,$1
li 0x0000,$2
li 0x0000,$3
MAC $1,$2,$3,0x0000,$6,$7
printr $3,$6,$7
print STRING_NEWLINE,$6,$7
li 0x01FE,$1
li 0x0000,$2
li 0x0000,$3
MAC $1,$2,$3,0x0001,$6,$7
printr $3,$6,$7
print STRING_NEWLINE,$6,$7
li 0xFEFE,$1
li 0x9595,$2
li 0x0000,$3
MAC $1,$2,$3,0x0000,$6,$7
printr $3,$6,$7
print STRING_NEWLINE,$6,$7
-23,0---Top- mactest.asm (WIMP Assembler) - Mon May  3 2:36PM - h:/PROJECT/s
```

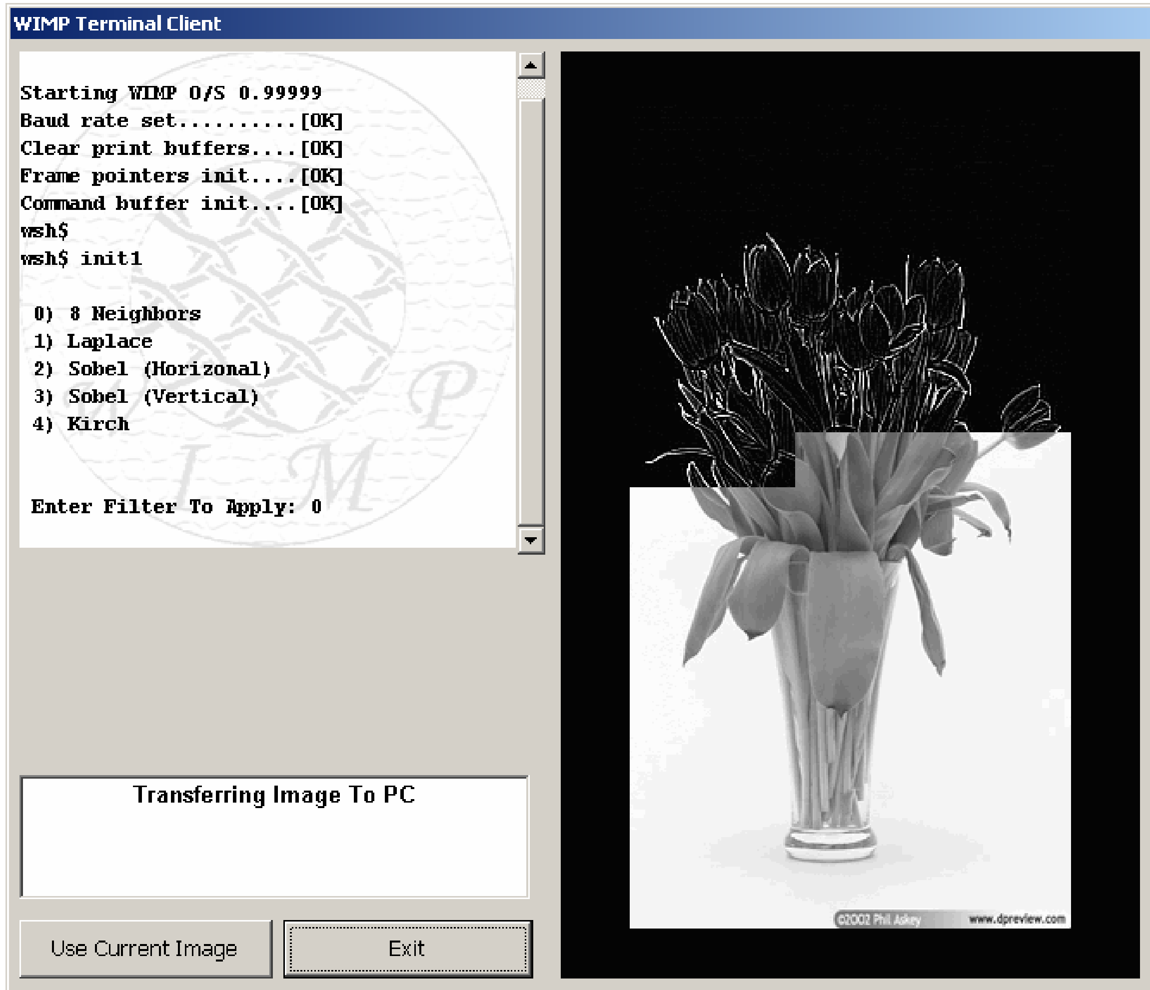
WIMPTERM (WIMP Terminal Client)

WIMPTERM is a specialized hyper-terminal application designed specifically for the WIMP processor. The program was written in MFC/C++ and therefore can be run on any Windows platform.

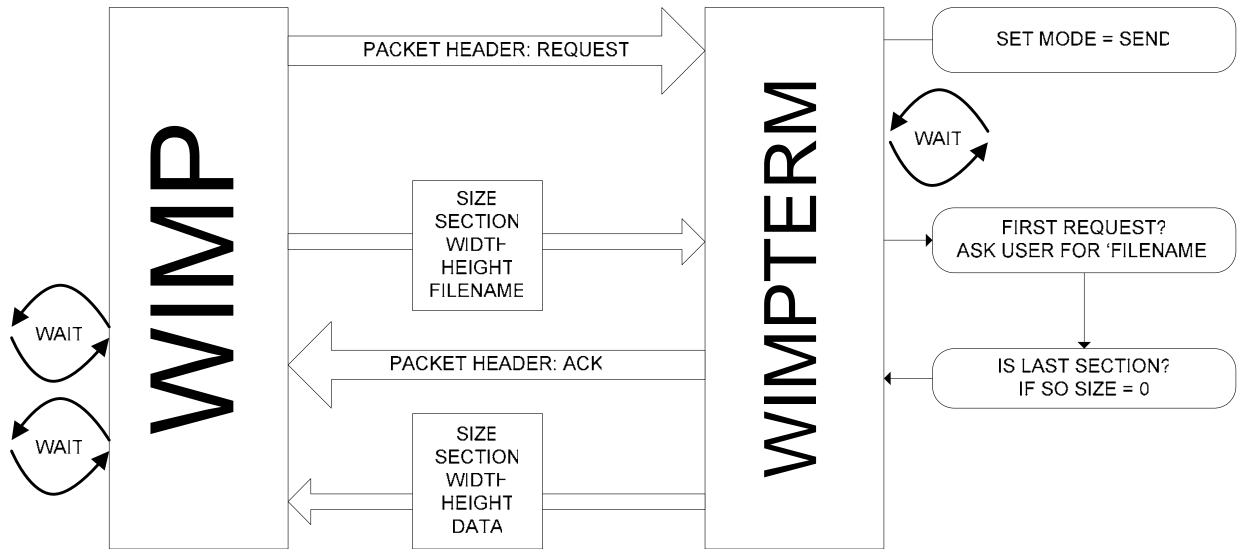
WIMPTERM easily allows the user to easily load images and binary finals into the processor. Below in a screenshot of WIMP waiting for the user to drop image into the lower-right dialog box. This dialog box also displays WIMPTERM status messages such as "Receiving Command", "Listening…", and "Transferring". Text in the upper left dialog box is data which has been sent from the SPAT. This dialog box also has been created to handle new page and backspace bytes as defined in ASCII.



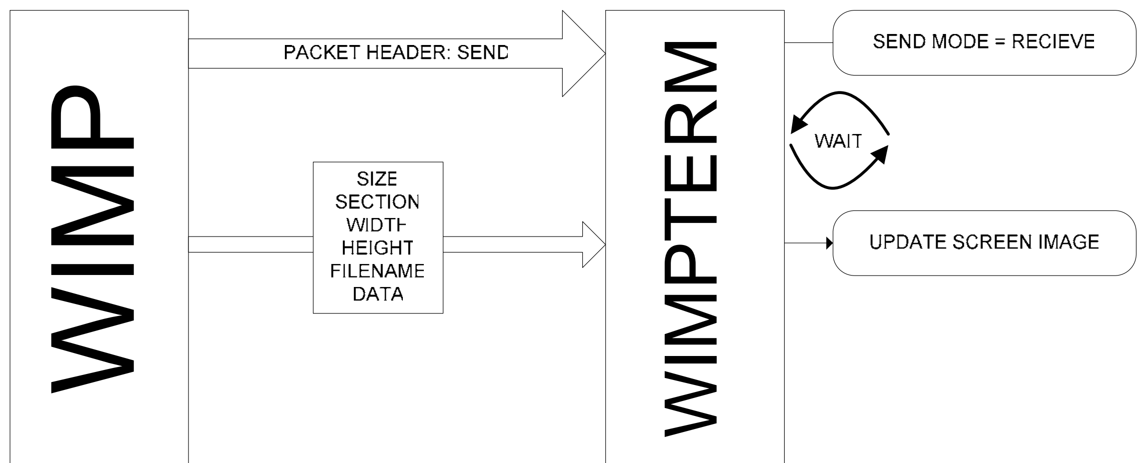
After an image is dropped into the into the box, WIMP and WIMPTERM transfer consecutive sections of the picture. In the image below, the upper half of the picture is data which has been received back from the processor and has been processed by an edge detection filter. The lower half of the image is unprocessed.



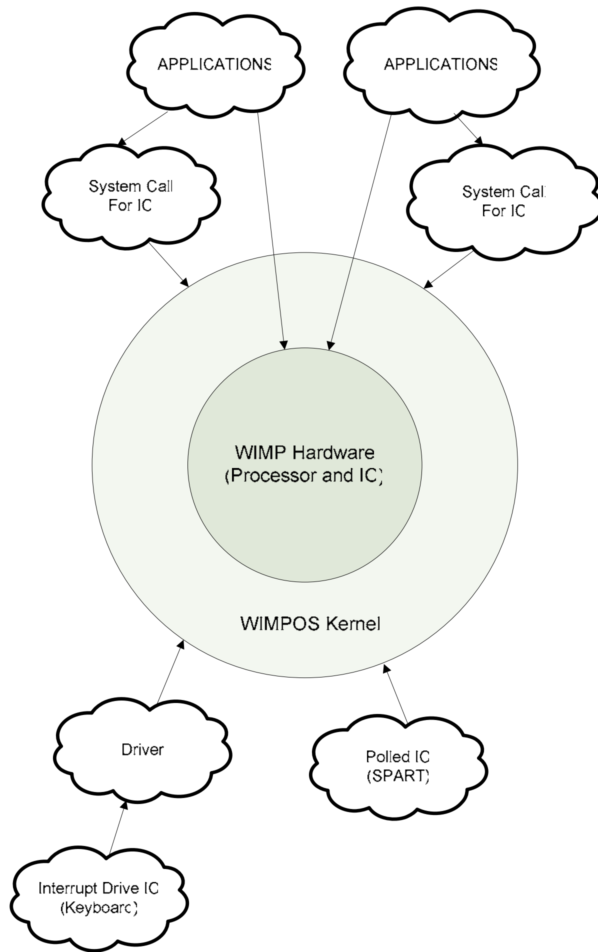
Protocol For Image Transmission: Receiving From PC



Protocol For Image Transmission: Sending From PC



WIMP Software Architecture



The heart of the WIMP system is the WIMP multi-threaded processor that interfaces all the hardware systems together. The WIMOS kernel manages the hardware and communicates between the system calls and the hardware. The system calls provide an interface between the applications and the WIMPOS kernel. They buffer the application requests that are serviced by the WIMPOS kernel. In addition, they also provide a command line interface to the user.

WIMPOS Kernel

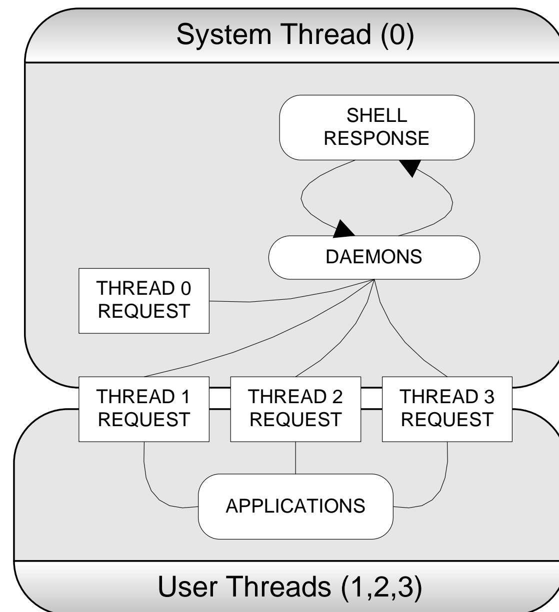
The WIMPOS kernel is a daemon that monitors all the buffers and takes the required actions to service the buffers. Applications running on different hardware can make the following requests through the system calls:

- Print to SPAT
- Get a data file from PC
- Send data to PC
- Get data from keyboard
- Load user program memory
- Load user data memory

In addition the WIMPOS kernel services the requests that come through the shell's command line interface, handle exceptions, and provide support for interrupts.

The kernel is actually a forever loop that samples all the buffers (listed in the memory map) and services them in a round robin scheme. The rest of the section discusses the buffer management and the command line interface provided by WIMPOS kernel.

I/O With Multiple Threads



Each thread has the ability to independently access system ports for input and output of data. When only one thread is running, there is no chance of simultaneous I/O requests, but when more than one thread is running, the I/O can interleave. For example, two threads attempt to print "AB" to the SPAR. Assuming they are started at the same time, the output to the hyper-terminal output will most likely be "AABB". These problems also occur with binary data trafers. To handle these cases, WIMPOS uses two daemons to handle string data and binary file transfers. The diagram to the right illustrates the construct of the daemons.

Printing to SPAT

An application in a particular thread writes into an out buffer if it requires an output on the screen. The OS then samples the buffer, if found non empty, it prints out the characters in the buffer and clears the buffer. One output buffer is provided for each thread. The buffer is byte addressable, e.g.

Print “ABCD”

A = address X
B = address X + 1
C = address X + 2
D = address X + 3

where X is the starting address of the buffer. A store into the buffer should be followed by a null byte being written into the buffer. (This use of null-delimited strings is common in many operating systems). The following table shows the address map for the print operation.

Table 23: Printing to SPAT				
	Thread 1	Thread 2	Thread 3	Thread 4
Starting Address	8000	8030	8060	8090
Ending Address	801F	804F	807F	80AF
Buffer Space	32	32	32	32
Buffer Status Address	8020	8050	8080	80B0

For example, say Thread 1 is running the OS kernel, and Thread 2 need to print ABCD. Thread 2 first checks if data in 8050 is 0. If it not, Thread 2 waits till 8050 is cleared. If so, ABCD is written into locations 8030, 8031, 8032 and 8033. Finally, thread 2 writes “4” into location 8050. The OS polls location 8050. If it is non empty (in this case 4), it writes out data into SPAT from 8030, 8031 and 8032 and 8033. It then clears 8050.

This function’s architecture does not support interleaving strings. For instance, if thread one wants to print “HELLO” and then “THERE” and thread two is simultaneous processing the same code, the output would be “HELLOHELLOTHERETHERE”. WIMPOS relies on the programs to intelligently output status to avoid this issue.

Reading from keyboard

The keyboard writes into an input buffer stream when ever a character arrives. In addition, it also writes into the output buffer steam for echo. The operating system then checks for the status of the buffer streams and takes the required actions.

Table 24: Reading from keyboard	
	Thread 1
Starting Address	8002
Ending Address	8021
Buffer Space	32
Buffer Status Address	8001

Example:

Say a key has been pressed. The keyboard driver checks the contents of 80E0. If that is non zero, the key is dropped. If it was zero, the keyboard driver writes the key into the input buffer (starting address 80C0) and the output buffer (starting address 8000, for an echo). It continues to do so till, an enter key has been pressed or 32 bytes have been entered. In either of the cases, the keyboard driver updates the address 80E0 (input buffer status) and 8020 (output buffer status).

The operating system polls 8001. If it finds that non-empty, it reads the buffer and clears the input buffer status.

Passing Parameters To Functions

In a system which has lines of execution and no direct memory addressing scheme, there is no way to support a single software stack. Therefore, when a thread calls a function it uses a dedicated area in the memory to send parameters to functions. This memory is a part of the system memory (8700 – 8FFF) and can be accessed in the program with the label .

FUNCTION_PARAM_T0 → Thread 0
FUNCTION_PARAM_T1 → Thread 1
FUNCTION_PARAM_T2 → Thread 2
FUNCTION_PARAM_T3 → Thread 3

The functions can return values to the calling function through the same set of memory locations. The assembler macros PASSUP and PASSDOWN automate the passing of values onto the parameter function stacks.

Getting a file from the PC

Applications are not restricted to the memory provided on the XESS board to store their data. Applications can request a file from the PC that is connected to WIMP through the serial port. WIMPOS acts as an interface between the application requesting the file and the PC. The following illustrates the process of requesting a file from the PC:

Application requests a file from the PC

The application running on a thread places the starting address of the file name, and starting address where the data in the file should be stored in the memory locations specified in WIMP's memory map. It then makes the buffer status as 1 indicating to WIMPOS that it has placed a request.

OS Services the request

WIMPOS comes around to the request, samples the buffer status and starts servicing the request. It calls the receive_file function that sends a request to the PC's monitor program for the file. Once the PC starts transmitting the file, the receive_file gets the file and places it in the memory location requested by the application. It then sets the buffer status to the value 2 indicating that the request has been completed.

Application acknowledges the completion of the request

The application then sets the buffer status to 0 indicating that it has acknowledged the completion and another request can be placed.

There is a buffer space provided for every thread. This way there is no contention for buffer space between the threads.

Sending a binary file to PC

The system supports sending a binary file to the PC through the serial port. This WIMPOS provides the interface between the application that needs to send the data and the serial port on the board. The following illustrates the process of sending a file to the PC:

Application generates a request to send the file to the PC

The application samples the buffer status (the addresses are provided in the memory map) to see if it is 0. If not, it waits till the buffer status is 0. Once the buffer status is 0, the application places the starting address of the binary file be sent, size of the binary file in bytes, and the starting address of the string that defines the file name. It then sets the buffer status to 1 indicating a request for file transfer.

WIMPOS services the request

Once WIMPOS checks the request, it services the request by calling the send_binary_file function. After the file has been successfully transmitted, the WIMOPS sets the buffer status back to 0 indicating the successful completion of the request.

Sending a file to the PC is a 2-step process unlike receiving a file that is 3-step process. There is no need for the application to wait for the data to be sent to the PC.

WIMP communicates with the PC through the serial port and the WIMPTERM application that runs on the PC. This interface allows:

- Loading data from PC to board
- Loading program from the PC to the board
- Sending a binary file from the PC to the board
- Sending a binary file from the board to the PC
- Sending a command to the PC

The command sent from WIMP to the PC begins with 0x05, 0x10, 0x11.

Applications requesting data from keyboard

Applications that need data from the keyboard call the system calls get_ch and get_string. These system calls in turn place requests on the buffer that is serviced by WIMPOS kernel. The following illustrates the process of getting an input from the keyboard:

Application calls the system call get_ch or get_string

The system calls wait on the input buffer.

Servicing shell commands

The commands that arrive from keyboard at the shell prompt are registered into the buffer (address present in WIMP's memory map) when the enter key is detected. When the kernel comes around and sees the buffer is non-empty, it reads the buffer and services the command. Once the command is serviced, it clears the buffer indicating that the next command can be placed into the buffer. Currently, commands arriving from the prompt are not queued. The following are valid commands used by the WIMP shell:

Command	Description
echo	Set keyboard echo-mode to 1
noecho	Set keyboard echo-mode to 0
cls	Send new page command to SPAT
init1	Start code at symbol THREAD_1
init2	Start code at symbol THREAD_2
init3	Start code at symbol THREAD_3
loadprog	Load user code/data from WIMPTERM
loadcode	Load user code from WIMPTERM
loaddata	Load user data from WIMPTERM

Updating The VGA

WIMP's four image processing programs require processing of image blocks because full images are larger than available memory. Because the VGA interface does not support a pixel addressing mode, the VGA RAM is treated as separately addressing linear arrays. The below look-up tables was using to locate the starting point of the array. The table is split between the two VGA memory segments. Using these addresses, 256 must be added to the current address to locate the next 'line' in the block image. The image blocks where chosen to be 32 x 32 bytes. This number was the greatest common divisor the height and width.

	0	1	2	3	4	5	6	7
0	0x0000	0x0020	0x0040	0x0060	0x0080	0x00A0	0x00C0	0x00E0
1	0x2000	0x2020	0x2040	0x2060	0x2080	0x20A0	0x20C0	0x20E0
2	0x4000	0x4020	0x4040	0x4060	0x4080	0x40A0	0x40C0	0x40E0
3	0x6000	0x6020	0x6040	0x6060	0x6080	0x60A0	0x60C0	0x60E0
4	0x8000	0x8020	0x8040	0x8060	0x8080	0x80A0	0x80C0	0x80E0
5	0xA000	0xA020	0xA040	0xA060	0xA080	0xA0A0	0xA0C0	0xA0E0
6	0xC000	0xC020	0xC040	0xC060	0xC080	0xC0A0	0xC0C0	0xC0E0
7	0xE000	0xE020	0xE040	0xE060	0xE080	0xE0A0	0xE0C0	0xE0E0
8	0x0000	0x0020	0x0040	0x0060	0x0080	0x00A0	0x00C0	0x00E0
9	0x2000	0x2020	0x2040	0x2060	0x2080	0x20A0	0x20C0	0x20E0
10	0x4000	0x4020	0x4040	0x4060	0x4080	0x40A0	0x40C0	0x40E0
11	0x6000	0x6020	0x6040	0x6060	0x6080	0x60A0	0x60C0	0x60E0
12	0x8000	0x8020	0x8040	0x8060	0x8080	0x80A0	0x80C0	0x80E0
13	0xA000	0xA020	0xA040	0xA060	0xA080	0xA0A0	0xA0C0	0xA0E0
14	0xC000	0xC020	0xC040	0xC060	0xC080	0xC0A0	0xC0C0	0xC0E0

Image Processing Applications

WIMP is designed to exploit the inherent parallelism present in multi-media applications. T0 is the system daemon that starts off an application in thread 1. Thread 1 initializes the application variables and then starts off thread 2 and thread 3. The three threads work on the image simultaneously. The threads synchronize and merge on regular basis, once every row of the image in this case.

Thresholding

This application reads an input image and compares every pixel to a threshold value. If the pixel value is greater than the threshold value, the pixel is made white, otherwise, it is made black. The user starts thread one (T1) from WIMPTERM to start the program. The application will print out eight different threshold values as options that can be used in comparing the pixels of the image. The getchar function will get the input to the WIMPTERM and will put the selected threshold value into memory. T1 will now initialize Thread 2 (T2) and Thread three (T3). T1 will send a request to the OS to get a block of the pixel, which contains 1024 bytes (32x32 pixels). Each thread will process two pixels at a time by comparing the pixel to the threshold value, and writing the result back into memory and VRAM. If T1 starts at address 0x9100, then T2 starts at 0x9002, and T3 starts at 0x9004. The program counter (PC) will be incremented by six each cycle so that the threads run through the image block correctly. At the end of the block, T2 and T3 will wait for T1 to send the processed block back to the computer and to VRAM. Then the image block is updated on the monitor using VGA and WIMPTERM. After the data is refreshed, the program will fetch another image block. This loop will continue until the image has been completely processed. Figure 3.1 below shows this process.

Threshold 0xAA ***Threshold 0x88*** ***Threshold 0x66***

START

Start
Thread 1

Fetch an Image
Block

Subword Parallel
Operations
1 word = 2 pixels

Start T3

Start T2

Process Pixel X+4
X+5

Process Pixel X
X+1

Process Pixel X+2
X+3

Block
Complete?

Y

N

Increment X by 6

Send Image Block

Wait and Merge with T1

Wait and Merge with T1

All Blocks Done?

N

Y

STOP

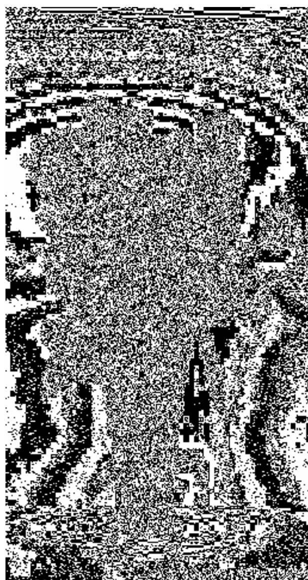
Figure 3.1: Bit Plane Slicing and Thresholding

Bit Plane Slicing

This application reads an input image and ANDS every pixel to a bit slice value. If the result is equal to zero, the pixel is made white; otherwise, it is made black. The following explains the mapping of the algorithm to the processor architecture:

Bit Slice	Bit Slice Value
Bit slice 0	0000 0001
Bit slice 1	0000 0010
Bit slice 2	0000 0100
Bit slice 3	0000 1000
Bit slice 4	0001 0000
Bit slice 5	0010 0000
Bit slice 6	0100 0000
Bit slice 7	1000 0000

The user starts thread one (T1) from WIMPTERM to start the program. The application will print out the eight different bit slice values as options that can be used to AND with the pixels of the image. Table 3.1 below shows all of the bit slice values that can be chosen. The getchar function will get the input to the WIMPTERM and will put the selected threshold value into memory. T1 will now initialize Thread 2 (T2) and Thread three (T3). T1 will send a request to the OS to get a block of the pixel, which contains 1024 bytes (32x32 pixels). Each thread will process two pixels at a time by comparing the pixel to the threshold value, and writing the result back into memory and VRAM. If T1 starts at address 0x9100, then T2 starts at 0x9002, and T3 starts at 0x9004. The program counter (PC) will be incremented by six each cycle so that the threads run through the image block correctly. At the end of the block, T2 and T3 will wait for T1 to send the processed block back to the computer and to VRAM. Then the image block is updated on the monitor using VGA and WIMPTERM. After the data is refreshed, the program will fetch another image block. This loop will continue until the image has been completely processed. Figure 3.1 above shows this process.



Bit Slice 0

Bit Slice 3

Bit Slice 6

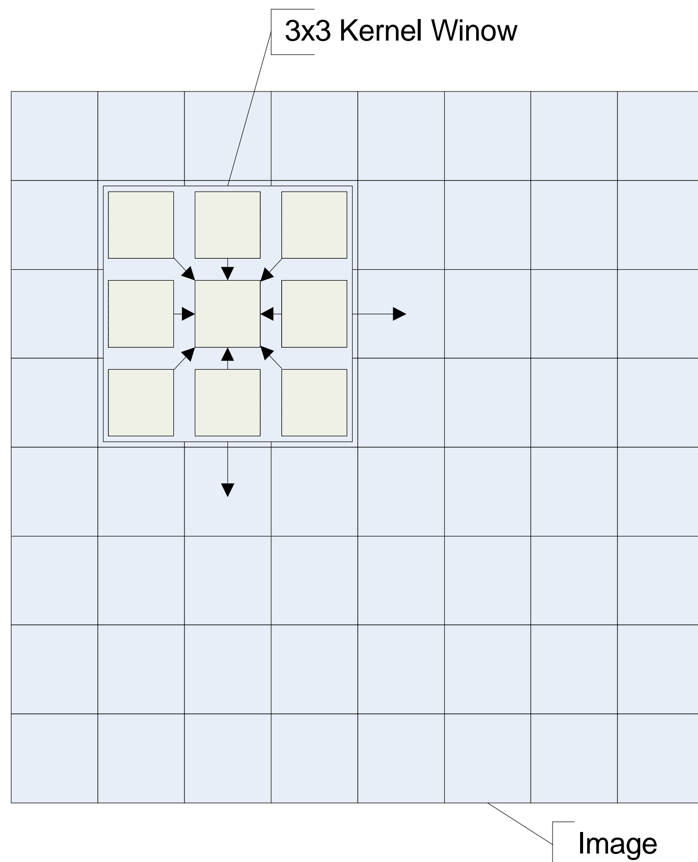
Edge Detection

The second application is a little more complex than image thresholding that involves filtering an image to detect the edges. This simple image filter sweeps a 3x3 kernel across the image replacing every pixel with a value that is the weighted sum of the neighboring pixels.

The application first pads 0's around the image to take care of the border cases. The illustration depicts the data input to the calculation of a single pixel. The kernel window translates across every pixel of the image and calculates a weighted sum between its values and the images' values which it covers. The sum is calculated in the following manner, given the image 'P' and kernel window 'K' have indexes relative to the window's center.

$$P(x, y) = \sum_{i=-1}^{1} \sum_{j=-1}^{1} K(i, j) P(x + i, y + j)$$

The new value of the pixel is stored in a new location so that it does not interfere with the other computations where the old value of 'P' needs to be used. The edge detection program has been created in such a way that any spatial filter can be applied by images. The current ones implemented have the following windows:



1	-2	1
-2	4	-2
1	-2	1

8 Neighbors

1	1	1
1	-8	1
1	1	1

Kirch

-3	-3	5
-3	0	5
-3	-3	5

Sobel Vertical

-1	-2	-1
0	0	0
1	2	1

Sobel Horizontal

-1	0	1
-2	0	2
-1	0	1

START

Start Thread 1

Compute Window Weight

Fetch an Image Block

Assign X=0, Y=1, Z=2

Assign Rows X, Y Z to different threads

Start T3

Start T2

Subword Parallel Operations
1 word = 2 pixels

Process Row X

Process Row Y

Process Row Z

Wait and Merge with T1

Wait and Merge with T1

Block Complete?

N

Increment X, Y, Z

Y

All Blocks Processed?

N

Y

STOP

Time Redundant Fault Tolerant Applications

Time redundant fault tolerant computing is another application domain that fits into the multithreading environment. The idea is to run the application that requires high fault tolerant standards on different threads and then compare the results. If the results are the same, the program execution was successful. On the other hand, if the results don't match, the program execution was not correct. The aim is just to detect the intermittent fault and re-execute the program when such a fault is detected.

WIMP supports fault tolerant computing as described in the micro-architectural manual. Stores are used as checkpoints in the program that requires fault tolerance. The instruction SWFT tracks the stores by writing the store values into a FIFO. The output ports 4 and 5 are shorted to input ports 4 and 5 respectively to allow a monitor program to track the and compare the stores. In addition, writing into output port 8 interrupts threads running the applications whenever required.

A simple scheme that requires the assistance of a compiler is to generate two copies of the application program where the store addresses are offset. The program may need to be profiled to know store addresses ahead of time. The two copies of the programs are executed on different threads completely before the monitor program takes over that checks all the stores that have been committed. If the monitor program finds a discrepancy, the application program is restarted. This scheme has a very slow response time since the programs have to be completed. Since WIMP supports tracking store values through the FIFO, this scheme is altered to decrease the response time to detect the fault.

In this application domain, we test two implementations of a time redundant fault tolerant scheme. The first scheme does not use ft support provided by the SWFT instruction. Only the multithreading capabilities of WIMP are used to make the program fault tolerant. In this scheme, two copies of the same program are executed with a small delay on two different threads. The programs use a modified form of store called "safe store" (a software wrapper around a regular store) as check points. The safe store writes the check point value into a defined memory location. The program blocks at this store till this checkpoint value has been read by the monitor program for comparison. Once the values have been read, the program continues regular operation till the next check point has been reached. This is a very slow implementation since the safe stores are blocking; and the program execution will not proceed until the check points have been read by the monitor program. If the values read are not the same, the monitor thread interrupts the program threads and restarts them. The general methodology using this scheme is shown in figure FT(A).

The second scheme uses the SWFT instruction that tracks the stores by writing the store values into the FIFO the same time store is committed. A second copy of the program is started with a latency not exceeding the FIFO depth, which is 256 in case of WIMP. The two copies write into the different FIFOs while a third thread continuously monitors the

FIFO's and compares the values. If the values popped from the two FIFO's are not the same, the thread interrupts the program threads and restarts them. This scheme executes faster since the check point stores are not blocking. Fig FT(B) shows the control flow in this implementation scheme.

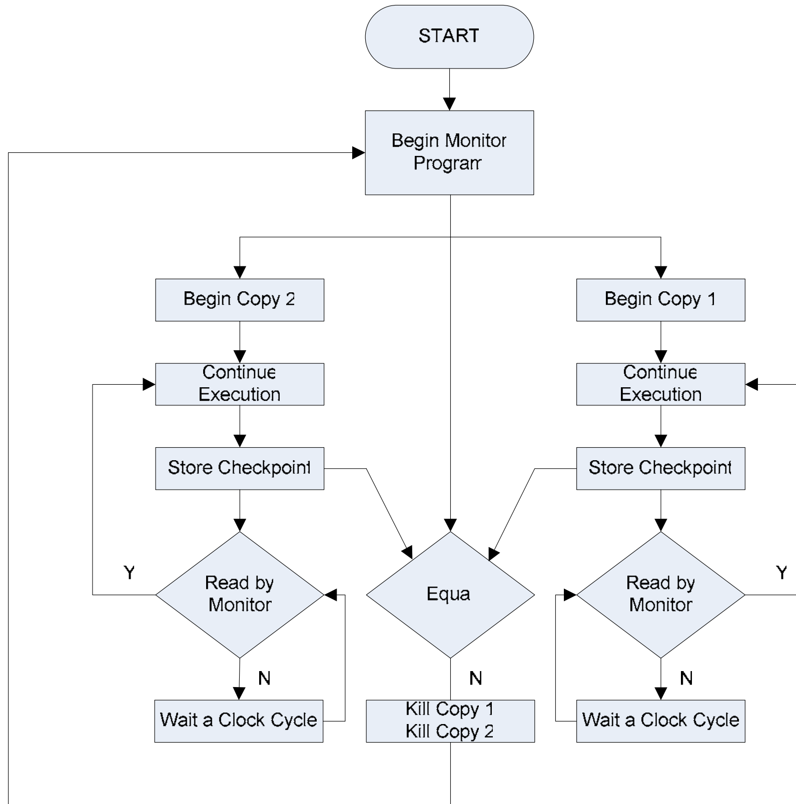


Fig. FT(A): Implementing fault tolerance without using SWFT

The fault tolerant scheme covers intermittent faults for

- Data flow
- Control flow

Data flow check is implemented easily using the schemes mentioned above. The same schemes can be used for control flow check by inducing the SWFT instruction at the end of every critical branch target. The SWFT should store/write a checksum that is computed using the previous control flow checkpoints.

A random number generator is used as a sample program to test the fault tolerant schemes in WIMP. A single fault is injected into the program using a global flag that is set to start off with. The flag is referred once to change a store value; and then it is cleared. The two copies are restarted following fault detection that find the global flag cleared. This time, the fault is not injected and the programs complete successfully.

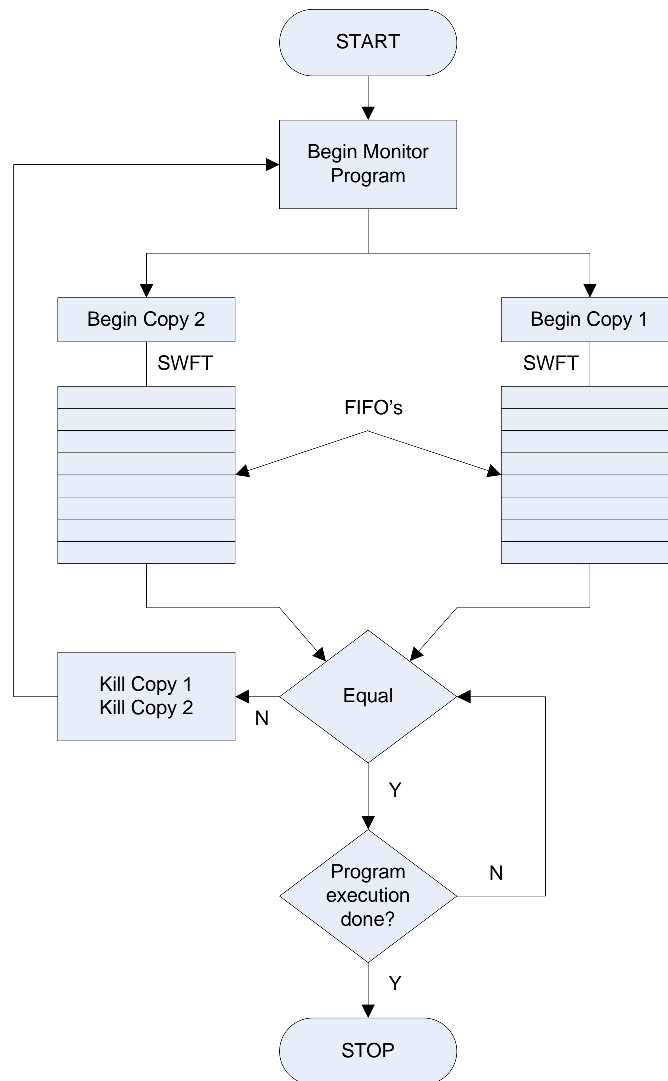


Fig FT(B): Implementing fault tolerance using SWFT

Following table shows the execution times of the program:

a) Without the fault tolerance scheme where the program is executed on a single thread
b) With the software fault tolerant scheme that does not use the SWFT support (SW Scheme)
c) With the fault tolerant scheme that uses the SWFT support (HW Scheme)

Program Length	SW Scheme	HW Scheme	Original Program
16 numbers	1124	983	293
256 Numbers	22724	2687	1805

Execution times of program

The numbers indicate the cycles taken by the program to execute the program. Thus this number is the actual time required to execute the program. As we can see, with little hardware support in the form of SWFT instruction, the overhead can be reduced to large extent. The numbers in the hardware scheme are also little bit larger as they include the error injection overhead. It is also evident that, as the program become larger and complex the overhead of monitor (comparator) program also becomes relatively less, as this overhead is constant and does not depend on length fault tolerant application.