Programmer's Manual for
XMTC Language, XMTC Compiler
and XMT Simulator
UMIACS-TR 2005-45 (Part 1 of 2)

Aydin O. Balkan and Uzi Vishkin

June 6, 2006
XMTC Version: **0.9**

Contents

Part I

Introduction

Chapter 1

The Purpose of this Manual

Explicit Multi-Threading (XMT) is a computing framework developed at the University of Maryland as part of a PRAM-on-chip vision (**http://www.umiacs.umd.edu/~vishkin/XMT**). Much in the same way that performance programming of standard computers relies on C language, XMT performance programming is done using an extension of C called XMTC.

The above mentioned web site provides a list of publications for readers interested in XMT Project. Two of these papers summarizes earlier research results and the first generation of the XMTC programming paradigm:

- U. Vishkin, S. Dascal, E. Berkovich and J. Nuzman. Explicit Multi-Threading (XMT) Bridging Models for Instruction Parallelism (Extended Summary and Working Document). Current version of UMIACS TR-98-05. First version: January 1998. (47 pages)

- D. Naishlos, J. Nuzman, C-W. Tseng, and U. Vishkin. Towards a First Vertical Prototyping of an Extremely Fine-Grained Parallel Programming Approach. TOCS 36, 5 pages 521-552, Springer-Verlag, 2003. (26 pages)

This manual presents the second generation of XMTC programming paradigm. It is intended to be used by an application programmer, who is new to XMTC. In the first part of this technical report (UMIACS-TR 2005-45 Part 1 of 2), we define and describe key concepts, list the limitations and restrictions, and give examples. The second part (UMIACS-TR 2005-45 Part 2 of 2) is a brief tutorial, and it demonstrates the basic programming concepts of XMTC language with examples and exercises.

Currently, there is no physical medium to execute a program written in XMTC. Source code is compiled and assembled properly, but the execution is simulated by an architectural simulator. This manual also explains how to use XMTC, given the current limitations of the simulator. In addition to the compiler and the simulator, our tool chain also includes a memory map creator to use with external data sets and a serializer for debugging.

Organization We divided this document into 5 parts. The following is a brief overview of the manual.

Part I Introduction

Chapter 1 presents the purpose of this manual.

Part II XMTC Language

Chapter 2 introduces new XMTC statements and explains their usage.

Chapter 3 describes new XMTC variable types.

Chapter 4 provides important information about function calls and available I/O methods.

Part III Simulation

Chapter 5 explains how to use the **memory map creator** for external datasets and memory allocation.

Chapter 6 explains how to use the **serializer** to test the program before running it with the simulator.

Chapter 7 explains how to use the **compiler** and **simulator** to compile and simulate the XMTC code.

Part IV Error and Warning Messages: This part lists and explains the warning and error messages that the compiler and the simulator may produce.

Part V Glossary

Chapter 9 shows the legend, and explains the XMTC related terms that we used in this manual.

Compatibility This document lists the features of XMT Tool Chain Version **0.9** as of **Feb-13, 2006**. While later versions are expected to be backwards compatible, there might be small changes. For up-to-date information on such changes, please consult to *XMTC web page*.

Related Documents This document is the first part of the UMIACS-TR 2005-45 technical report. The second part focuses on programming examples using XMTC language. The most up-to-date version of these documents can be accessed at `http://www.umiacs.umd.edu/users/vishkin/XMT` web site.

History The initial version of this document is completed on February 2005. Based on the improvements in the XMT Toolchain, some sections have been revised on May 2005 and February 2006.

Acknowledgments We would like to thank the current members of XMT Research Team for the contributions to the content of this manual, and their valuable comments.

Contact Person Aydin Balkan: balkanay@umd.edu

XMTC Web Page http://www.umiacs.umd.edu/~balkanay/xmt/

Part II

XMTC language

Chapter 2

New Statements of XMTC

The XMTC language is a superset of the C programming language. Ideally, a serial program written in pure C would compile and run through the tool chain without any problems. This chapter summarizes the new statements of XMTC language that allow parallel programming in a PRAM-like style.

2.1 spawn

2.1.1 Usage

```
spawn(start_thread, end_thread)
{
  // Spawn Block :  insert parallel thread code here
}
```

This statement spawns $end\_thread - start\_thread + 1$ virtual threads, which concurrently and independently execute the code block following the instruction. The threads assume the ID numbers within the inclusive interval $[start\_thread; end\_thread]$. In the *Spawn Block* the *Thread ID* can be accessed using the symbol $.

The XMT processor is said to be in *Parallel Mode* during the execution of the *Spawn Block*.

2.1.2 Requirements and Restrictions

1. **Parameters:** Both of the parameters **start_thread** and **end_thread** must be integer expressions. The expressions must not have a comma.

2. **Variable Declaration:** New variables that are used within the *spawn block* can be declared at the top of the *spawn block*. Such variables will be private to each *Virtual Thread*, i.e. their values may vary from one *Virtual Thread* to another *Virtual Thread*.

3. **Thread IDs:** The first spawned thread will assume the value of **start_thread** as the *thread ID*, the last thread will assume the value of **end_thread** as thread ID (start and end IDs are inclusive).

4. **Function Calls:** Function calls (both system and user functions) are not allowed from within the *Spawn Block*.

8

5. **Access to Variables:** All variables declared within the spawn block are fully accessible, i.e. they can be read or modified freely. Variables declared with type `psBaseReg` can not be accessed within a spawn block directly. They must be read or modified through a `ps` statement.

6. **Concurrent Writes:** Concurrent writes within a *Spawn Block* are not checked by the compiler. Such situations may result in incorrect execution. The user must be aware of possible concurrent write situations and avoid them (Example 2).

7. **Pointers:** Pointer arithmetic is not allowed within the *Spawn Block*. This is not checked by the compiler.

8. **Nesting:** `spawn` statements **cannot be nested**. In order to spawn more threads from within the *Spawn Block* (during parallel mode), use `sspawn` statement.

9. **Number of Instructions:** If the number of (assembly) instructions within the *Spawn Block* exceeds **300**, the compiler will issue a warning and continue compilation. Assuming that everything else is correct, your program will compile and simulate regardless of instruction count. However, your results may not be correct. If you see such a warning, please inform the XMT Research Team.

2.1.3 Examples

Example-1 An example of legal `spawn` usage

In the following code we spawn *virtual thread*s with *thread IDs* ranging from 0 to $N - 1$. Then we create an integer variable `temp`. Each *virtual thread* has its own copy of this variable. We read the array `C` using an expression containing the `$` character. Based on this value, we copy an element from `B` array to `A` array either as it is, or the negative of it.

```
int main()
{
  ...
  int N;
  ...
  spawn(0, N-1);
  {
    int temp;
    temp = C[$*2];
    if(temp > 0) {
      A[$]=B[$];
    } else {
      A[$]=-B[$];
    }
  }
  ...
```

Example-2 Writing some value to a variable by multiple virtual threads

According to Arbitrary-CRCW PRAM model, if multiple threads attempt to write to a memory location, an arbitrary one of them will succeed. This arbitration needs to be handled by the programmer. Otherwise the outcome of the program may be undetermined.

For consistency, prefix-sum statements (`ps` or `psm`) must be used to execute concurrent writes These statements act as gatekeepers that allow only one of the writers to go through. See Section 2.3 for more details on these statements.

```
Incorrect handling of variable j:

int j; // j is a global variable

int main()
{
  // Beginning of a serial block

  // Some Serial Code

  spawn(low, high)
  {

    int temp;

    ...

    temp = ...

    ...

    j = temp;

    ...
  }

  // Rest of the Program

}
```

```
Correct handling of variable j:

psBaseReg gateKeeper;

int j;

int main()
{
  //Beginning of a serial block

  // Some Serial Code

  gateKeeper = 0;

  spawn(low, high)
  {
    int i;
    int temp;
    ...
    i=1;
    ps(i,gateKeeper);

    // Only one thread will get
    // 0 from ps instruction
    if(i == 0) {
      j = temp;
    }

    ...
  }

  // Rest of the Program
}
```

In the code on the left, `j = temp` assignment is a concurrent-write operation. In this case, different threads may write different values to `j` in some arbitrary order. The order of writing may depend on both hardware and software components and it is not predictable. The final value of `j` and the identity of the modifying thread can not be determined after the spawn block.

In the code on the right, the `j = temp` assignment is encapsulated by a gatekeeper operation. The atomic `ps` operation ensures that only one thread gets permission to modify `j`. In this case both the value and the identity of the modifying thread can be determined after the spawn block.

2.2   Single Spawn (`sspawn`)

2.2.1   Usage

```
spawn(low, high)
{
  int child_ID;

  // Some parallel code here

  sspawn(child_ID)
  {
    // Initialization Block:
    // Code for initializing the child thread
  }
  // Some other parallel code here
}
```

With this statement, the current thread (*parent thread*) spawns **a single** *virtual thread* (*child thread*). The parent thread executes the code in the *initialization block* that follows the statement, then the *child thread* starts from the first line of parent's *spawn block*. The *thread ID* of the newly spawned thread is copied to the parameter of the `sspawn` (`child_ID`). This value can be read from within the *initialization block*.

2.2.2   Requirements and Restrictions

1. **Parameter:** The parameter must be an integer declared within the parent's *spawn block*. Constant numbers and mathematical operators cannot be used as a part of the parameter.

2. **Thread ID:** Within the *initialization block* the character $ still refers to the *thread ID* of the *parent thread*.

3. **New Variables:** In an *initialization block* new variables may not be declared. This is to avoid concurrent writes into the stack in the shared memory.

4. **Synchronization:** In order to prevent premature starting of child threads, a synchronization mechanism must be used (Example 4).

2.2.3   Warnings

1. Every time the `sspawn` statement is encountered, one more thread will be spawned. Since this new thread executes the same *spawn block*, it may encounter the `sspawn` statement at some point during the execution. This feature allows recently spawned threads to spawn more threads, as opposed to limiting the spawning ability to the original set of threads. On the other hand, if `sspawn` is not used correctly, an infinite number of threads might be spawned. Therefore, this statement should be used with caution, and preferably enclosed within a control structure such as `if` or `while` (or others) (Example 3).

2. The value in `child_ID` prior to the execution of this statement will be overwritten.

2.2.4   Examples

Example-3   Avoiding infinite spawns

The following example shows a method of avoiding infinite spawns. Please note that the method shown in this example may not be the most efficient one in terms of program performance. In order to reduce complexity, the synchronization statements are not shown in the below program

```
Uncontrolled sspawn leads to infinitely
many threads (and eventually, crashing)

int N; // Total amount of work
int M; // Initial thread count
       // N >> M

int main()
{
  // Beginning of a serial block

  // Some Serial Code

  spawn(0, M-1)
  {
    int childID;

    sspawn(childID)
    {
      // Initialize child here
    }

    if( $ < N )
    {
      // do some work
    }
  }

  // Rest of the Program

}
```

```
Controlled sspawn statement spawns as
many threads as needed

int N; // Total amount of work
int M; // Initial thread count
       // N >> M

int main()
{
  // Beginning of a serial block

  // Some Serial Code

  spawn(0, M-1)
  {
    int child_ID;
    int my_workload;
    if($<M)
    {
      my_workload=N/M;
    }
    else
    {
      my_ workload=1;
    }


    while(my_workload>1)
    {
      sspawn(child_ID)
      {
        // Initialize child here
      }
      my_workload--;
    }

    // do some work

  }
  // Rest of the Program
}
```

12

Example-4   Synchronization for `sspawn` statement

The following example uses `volatile` variables as locks (or semaphores) to prevent the child thread from starting before its initialization is completed.

```
volatile int locks[100];
...
int main()
{
  ...
  spawn(low,high)
  {
    int child_ID;
    if (thread is a single-spawned thread) {
      while (locks[$]==0); // spin-wait
    }
    ...
    sspawn(child_ID)
    {
      ... Initialization Block: Code for newly spawned thread
      locks[child_ID]=1; // give signal to child
    }
    ... Some other parallel code here ...
  }
  ... Some other serial code here ...
}
```

2.3 Prefix Sum (ps), and Prefix Sum to Memory (psm)

2.3.1 Usage

```
ps(int local_integer, psBaseReg ps_base);
```

```
psm(int local_integer, int* p_variable);
```

Both statements execute the following operations **atomically**:

- Add the value of the local_integer to the second parameter (ps_base for ps, memory location for psm).

- Copy the old value of the second parameter to the local_integer

2.3.2 Requirements and Restrictions

1. **Parameters:** local_integer must be declared as int **within the current *Spawn Block*.** Parameter ps_base must be declared **globally** as psBaseReg. Parameter p_variable must be a pointer to an integer variable. See Chapter 3 for details. Also note that, constant numbers and mathematical operators cannot be used as a part of these parameters.

2. For ps statement the value of local_integer must be equal to 0 or 1 prior to the execution. There is no such restriction for psm statement.

3. **Number of prefix-sum base registers:** Due to architectural constraints, the user can only declare 8 variables of type psBaseReg and use in ps statement. In future versions, this restriction may be relaxed.

2.3.3 Warnings

1. **Performance:** In performance applications, the programmer should opt for ps statement over psm statement, whenever possible. The XMT microarchitecture provides a more performance-efficient execution for ps statement. ps statement from different threads are executed concurrently, while the psm operation of different threads may be queued. In terms of performance, this feature is analogous to working on a variable in architectural registers as opposed to working on the same variable in main memory without loading it into registers.

2.3.4 Future Extensions

A feature that is still under development is the mark(int i, int *p_variable); statement. This statement will write the value of i to the address pointed by p_variable, in a way that psm statement does. The difference is that the old value of the location pointed by p_variable will not be copied back to integer i.

Chapter 3

Variables

3.1 Local and Global Variables of XMTC

XMTC adheres to the *global variable* definition of the C language: A *global variable* can be accessed from every point in the code.

We will examine *local variable*s in two groups:

1. C-like local variables: Variables in this group are treated the same as in C language. A *local variable* can only be accessed from within the scope that it is declared (e.g. a function block).

2. Thread-local variables declared within a *spawn block*: Such variables are private to each *virtual thread*. In other words, each *virtual thread* has its own copy of that variable.The first parameters of `ps` and `psm` statements as well as the parameter of `sspawn` statement must be integers of this type. They must be declared at the beginning of the *Spawn Block*.

Currently, XMTC allows the use of following types of variables:

- character (`char`)

- integer (`int` and `long`)

- single/double precision floating point (`float` and `double`)

- above types with `const`, `static` and `volatile` modifiers

- arrays of above types

- `struct`s, and arrays of `struct`s containing above types

- pointers to above types

`typedef`, `union`, and `enum` keywords are also allowed in variable declarations. Their usage is the same as in C language.

Currently, `int` and `double` variables are represented in 64-bits.

3.1.1 Requirements and Restrictions

1. Variables must not contain the character `$`.

2. `unsigned` is not supported.

C-like Local Variables

1. The user cannot use integer arrays with 64-bit values as part of their initialization.

   For example, the following would be incorrect:

```
main() {
  int foo[4] = {
       0xffffffffffffffff,
       0xffffffff00000000,
       0xffff000000000000,
       0xff00000000000000
     };
  // Rest of the main()
  ...
}
```

   This applies to both local arrays and global arrays that are initialized as part of the source code. Global arrays can be initialized externally using the memory map tool.

Local Variables within Spawn Block

1. The local variables that are declared within the *spawn block* are stored in the architectural registers of the *TCU*. Since there are a limited number of hardware resources to be used as for this purpose, the programmer should be conservative in declaring such variables.

2. New variables may not be declared within the *Initialization Block* after `sspawn` instruction.

3.2   New types of variables

psBaseReg Variables of this type are base variables for the prefix-sum operation. A `ps` statement must use a variable of this type as the second parameter.

3.2.1   Requirements and Restrictions

psBaseReg

1. **Declaration:** These variable must be declared globally, before the main() function.

2. **Variable Access:** Variables of type psBaseReg are accessible (can be read/written) normally from the *serial section*. From the *parallel section* they can be accessed **only** by `ps` statement.

3. **Structural Limitation:** Due to architectural limitations, at most 8 variables of this type can be declared in a program. In the future versions this limitation may be relaxed.

Chapter 4

Functions and System Calls

Function calls of any kind are not allowed in the *Spawn Block*.

4.1 User Defined Functions

XMTC supports user defined functions. Programmers can write functions following the rules and constraints of the C language.

4.2 System Calls

Currently, the XMTC simulator does not support calls to the operating system. In the future, XMTC will support frequently used libraries, such as I/O operations and math library. Currently, programmer must refrain from using system calls and libraries.

4.3 Available Input/Output Methods

4.3.1 Data Input

The programmers can use the Memory Tool (see Chapter 5) in order to prepare their external data to use with the simulator.

4.3.2 Data Output

printf Statement

The programmers can use `printf` statement in order to print program results to the terminal window (standart output or `stdout`). The usage of the `printf` statement is similar, yet not identical, to the printf function of the `stdio` system library.

4.3.3 Requirements and Restrictions

1. The `#include <stdio.h>` compiler directive is not required.

2. At most 3 parameters can be used in the `printf` statement.

3. At most 1 double value in the `printf` statement.

4. The format string supports only basic format characters. Width modifying strings such as "`%02d`" are not supported.

Part III

Simulation

Chapter 5

External Datasets and Memory Allocation

5.1 Overview

Currently, XMT simulator is not able to handle calls to the operating system. Two types of frequently used system calls are file operations such as reading the input data from a file, and dynamic memory allocation. A temporary method has been developed for the users to work with external data, and allow them to allocate memory statically:

1. The user identifies the data structures present in an XMT program, and prepares *content files* in appropriate format.

2. Using `memMapCreator` tool (*Memory Tool*) with these *content files* the user prepares

    (a) A header file to be used with the program code (*Memory Map - header file*)

    (b) A binary file to be used as the input data file of the simulator (*Memory Map - binary file*)

    (c) As a byproduct, the tool also generates a text file showing the contents of the binary file (*Memory Map - text file*). This file is not being used by the C code nor the Simulator. It may be used for testing or debugging purposes.

3. The header file has to be included (either using `#include` directive or `-include` compiler option) in the program code. (see Section 7.2)

4. The binary file has to be loaded to the simulator using the `-binload` option (see Section 7.5)

5.2 Identifying Data Structures

5.2.1 External Data

The first task of preparing the external input set is identifying the data structures to be used in the program.

- The user has to initialize the *scalar variable*s, and *array variable*s in the memory.

- Currently the *memory tool* can be used to work with **64-bit scalar integer/double variables** and **1 or 2 dimensional 64-bit integer/double arrays**.

- For each variable, the user is required to

  1. Declare its name
  2. Declare its (dimensions and) size
  3. Choose the content of each variable among
     (a) 0 (for scalar variables)
     (b) A fixed value (for scalar variables)
     (c) All elements 0 (for arrays)
     (d) All elements uniformly random between 0 and 1 for floating point arrays, and between 0 and a user defined upper limit for integer arrays
     (e) A text file (*content file*) containing the value of each element (for arrays)

5.2.2 Static Memory Allocation

The user can use the above method for allocating portions of the memory to be used in the program. The size of these portions needs to be known a priori. Allocation can be made using arrays. For example, 1024 words of memory for integer values can be allocated by creating the array `int temp1024[1024];` using the above method. Later they can be accessed similar to regular arrays.

5.3 Using the Memory Tool `memMapCreate`

5.3.1 Introduction

This tool is designed to help with creating header and binary files to be used with the XMT compiler/simulator toolchain. In order to navigate within the program enter the number or letter or symbol for the desired action and hit *Enter*. Here we are describing **Revision 1.3** of this tool. The revision number is displayed above the main menu as the program is started.

5.3.2   Main Menu

```
****************************************************
****************************************************
*                                                  *
*            XMTC Header File Creator              *
*                  Revision 1.3                    *
*                                                  *
*                                                  *
*                M A I N   M E N U                 *
*                                                  *
*  1. Set File Names                               *
*  2. Read/Write Header and Memory Map             *
*  3. Set Random Number Seed                       *
*  q. Quit                                         *
*                                                  *
****************************************************


 *** > _
```

1 **Set File Names** Takes you to the *Set File Names* Menu (Section 5.3.3)

2 **Read/Write Header and Memory Map** Takes you to the *Read/Write Files* Menu (Section 5.3.4)

3 **Set Random Number Seed** Displays you the default random number seed, and asks you if you want to change it. If you answer with **y** the program asks you for a new seed.

q **Quit** Quits the program

5.3.3   Set File Names Menu

```
****************************************************
****************************************************
*                                                  *
*            S E T   F I L E   N A M E S           *
*                                                  *
*  1. Set Header Name                              *
*  2. Set Memory Map Name                          *
*  3. Set Text file Name                           *
*  <  Back to previous Menu                        *
*                                                  *
****************************************************


 *** > _
```

1 **Set Header Name**

- Displays the current name for the header file, and asks you for a new one.
- You must type one character at least, before hitting *Enter*.

22

- The program does not add the file extension `.h` by itself, you need to type it explicitly.
- You must enter a valid name here before executing **R** or **H** commands in the *Read/Write Files* Menu (Section 5.3.4)

2 Set Memory Map Name

- Displays the current name for the **binary Memory Map** file, and asks you for a new one.
- You must type one character at least, before hitting *Enter*.
- The program does not add the file extension `.bin` by itself, you need to type it explicitly.
- You must enter a valid name here before executing **B** command in the *Read/Write Files* Menu (Section 5.3.4)

3 Set Text File Name

- Displays the current name for the **ASCII Memory Map** file, and asks you for a new one.
- You must type one character at least, before hitting *Enter*.
- The program does not add the file extension `.txt` by itself, you need to type it explicitly.
- You must enter a valid name here before executing **B** command in the *Read/Write Files* Menu (Section 5.3.4)

< **Back to previous Menu** Goes Back to the *Main Menu* (Section 5.3.2)

5.3.4   Read / Write Files Menu

```
****************************************************
****************************************************
*                                                  *
*         R E A D / W R I T E   F I L E S          *
*                                                  *
*  1. Add Integer Scalar Variable                  *
*  2. Add Integer Array Variable                   *
*  3. Add Double Scalar Variable                   *
*  4. Add Double Array Variable                    *
*  R. Read Variables from Header File              *
*  L. List Current Variables                       *
*  D. Delete Last Variable                         *
*  H  Create Header File                           *
*  B  Create Text and Binary Files from sources    *
*  <  Back to previous Menu                        *
*                                                  *
****************************************************

 *** > _
```

1 Add Integer Scalar Variable

- Asks the name of the scalar variable.
- Confirms the name.
- Asks for the Value. You must enter one digit at least. During creation of the text/binary file, if the entry is text (not a number), it will be converted to 0. If the entry is a floating point number, it will be rounded down. (uses `atol()` function of C).
- **Warning**: The program does not check for identical variable names. The programmer is responsible to track the names of the variables.

2 Add Integer Array Variable

- Asks the name of the array variable.
- Asks the dimension of the array variable. Currently you can only create 1 or 2 dimensional arrays.
- Asks the sizes of each dimension. The size of each dimension will be added as a scalar variable. For example, for the array called `myArray[1024]` there will be a scalar integer variable `myArray_dim0_size`, which has the value 1024. If the array would be two dimensional, such as `array2D[10][20]`, there will be two scalar integer variables: `array2D_dim0_size` with the value 10, and `array2D_dim1_size` with the value 20.
- Asks for the source. You have 3 options:
  (a) <file> : Reads one integer from the *content file* <file> per element. If the variable is two-dimensional, the second dimension is read first, i.e. the elements `array[0][0]` to `array[0][array_dim1_size - 1]` are read first. The program reads only as many elements as the array contains (for example, 15 elements for `array[3][5]`). If the *content file* has more elements, they will not be read.
  (b) 0 : Sets all elements to 0
  (c) R : This is for filling the array with random elements. The program asks for the upper bound. The lower bound is always 0. If the upper bound is entered as 0, the default value of 10^6 replaces 0.
- **Warning**: The program does not check for identical variable names. The programmer is responsible to track the names of the variables.

3 Add Double Scalar Variable

- Asks the name of the scalar variable.
- Confirms the name.
- Asks for the Value. You must enter one digit at least. During creation of the text/binary file, if the entry is text (not a number), it will be converted to 0. The `atof()` function of C standart library is used for conversion.
- **Warning**: The program does not check for identical variable names. The programmer is responsible to track the names of the variables.

4 Add Double Array Variable

- Asks the name of the array variable.
- Asks the dimension of the array variable. Currently you can only create 1 or 2 dimensional arrays.

- Asks the sizes of each dimension. The size of each dimension will be added as a scalar integer variable. For example, for the array called `myArray[1024]` there will be a scalar integer variable `myArray_dim0_size`, which has the value 1024. If the array would be two dimensional, such as `array2D[10][20]`, there will be two scalar integer variables: `array2D_dim0_size` with the value 10, and `array2D_dim1_size` with the value 20.

- Asks for the source. You have 3 options:

  (a) <file> : Reads one double precision floating point number from the *content file* <file> per element. If the variable is two-dimensional, the second dimension is read first, i.e. the elements `array[0][0]` to `array[0][array_dim1_size - 1]` are read first. The program reads only as many elements as the array contains (for example, 15 elements for `array[3][5]`). If the *content file* has more elements, they will not be read.

  (b) 0 : Sets all elements to 0

  (c) R : This is for filling the array with random double precision floating point numbers between 0 and 1.

- **Warning**: The program does not check for identical variable names. The programmer is responsible to track the names of the variables.

R **Read Variables from Header File**

- Reads the header file with the name declared in *Set File Names* Menu (Section 5.3.3).

- The header file must be created previously by this program using **H** command of this menu. Otherwise the program may not recognize the variables. (Manually modifying the header file is possible, yet strongly discouraged)

- The read variables are **added** to current list. If you read the same header file twice, all variables will appear twice, which may cause the compiler to throw errors because of duplicate definition.

L **List Current Variables**

- Lists current variables in the memory that are either read from the header file using **R** command, or created using **1, 2, 3** or **4** commands. An example screenshot is below:

```
 *** > L
Name       : gen_array
Dimension : 1
Size [0]   : 1024
Source     : R 10000

Name       : gen_aux_array
Dimension : 1
Size [0]   : 1024
Source     : 0

Name       : gen_temp_array
Dimension : 1
Size [0]   : 1024
Source     : 0

Name       : gen_pointer
Dimension : 1
Size [0]   : 1
Source     : 0

Name       : gen_randomNumbers
Dimension : 1
Size [0]   : 500
Source     : R 65536
```

D **Delete Last Variable**

- Deletes the last variable at the end of the list shown by **L** command.
- This action is not undoable.

H **Create Header File**

- Creates the header file with the name defined by **1** command in *Set File Names* Menu. (Section 5.3.3)
- If there is already a file by that name, it will be overwritten without a notice.

B **Create Text and Binary Files from sources**

- Creates text and binary files with the names defined by **2** and **3** commands in *Set File Names* Menu. (Section 5.3.3)
- If there are already files by these names, they will be overwritten without a notice.
- The files defined as sources of array variables (see **2** command of this menu) must exist.

< **Back to previous Menu** Goes Back to the *Main Menu* (Section 5.3.2)

5.3.5   Using Input Files

The keystrokes for generating a particular set of memory files (header and binary files) using the memMapCreate program can be externally stored in a text file. Such a text file can be fed into the memMapCreate program using basic redirection operators.

Example-5   Using an input file with the memMapCreate program

The following input file does the followings:

- Set the header file name to `myHeader.h`
- Set the binary data file name to `myData.bin`
- Set the text data file name to `myData.txt`
- Create an integer scalar variable with name `a` and value 50
- Create an integer scalar variable with name `b` and value 100
- Create an integer one-dimensional array with name `arr1` and size 500. The contents of the array will be read from the text file `array1.txt`.
- Create an integer one-dimensional array with name `temp1k1` and size 1024. All elements of this array will be equal to 0.
- Create header and data files
- Quit memMapCreate

Suppose that this input file is saved with the name `inputFile.txt`. To use this file with memMapCreate program, type:

```
memMapCreate < inputFile.txt
```

Contents of the input file `inputFile.txt`

```
1
1
myHeader.h
2
myData.bin
3
myData.txt
<
2
1
a
y
50
1
b
y
100
2
arr1
1
500
array1.txt
y
2
temp1k1
1
1024
0
y
h
2
b
<
q
```

5.3.6 Known Bugs

We would appreciate, if you inform us in case you encounter the following bug or new bugs.

1. If the backspace key is used under certain conditions the program will encounter a segmentation-fault when writing out the files.

5.4 Using the Generated Header File

The generated header file includes the declaration for all the global variables that must be initiated in the shared memory before the execution of the simulator is started, as well as *size* variables for

the array variables. Additionally, the header file may also contain the declaration for the temporary arrays. An example is below:

```
int degrees[1000];
int degrees_dim0_size;

int edges[10000][2];
int edges_dim0_size;
int edges_dim1_size;
```

Here, the user requested the 1-D array `int degrees[1000]` and the 2-D array `int edges[10000][2]` by using the memMapCreate program described in Section 5.3. The additional scalar variable `int degrees_dim0_size` has the value 1000, and the variables `int edges_dim0_size` and `int edges_dim1_size` have the values 10000 and 2 respectively.

The header file must be included during the compilation of the XMTC file either by using the `#include` directive similar to including regular C header files, or the `-include` option of the compiler. For more details please see Section 7.2.

5.5   Using the Generated Binary File

The generated binary file could be used with the XMT simulator using the `-binload` option. This option will load the binary file into the simulator's memory before the simulation begins.

Alternatively, the text file that is generated by the memMapCreate file can also be loaded to the simulator using the `-textload` option. This alternative is provided for convenience of the user. The performance or results of simulation that uses the text file is not different compared to a simulation that uses the binary file. For more details please see Section 7.5.

Chapter 6

Testing the XMTC Program Before Simulation

6.1 XMTC Serializer

6.1.1 Introduction

The rest of this section is copied from the documentation of the XMTC Serializer. This program with full documentation can be obtained from http://www.umiacs.umd.edu/~balkanay/xmt/.

In this section, the terms 'user' and 'programmer' refer to the reader. 'XMTC Serializer' and 'Serializer' are used interchangeably, and refer to the program that is meant for the user.

6.1.2 Motivation

The XMT toolchain consists of:

1. XMTC Compiler

2. Assembly compiler

3. Assembler

4. XMT Simulator

Problem   Both the XMTC Compiler and the XMT Simulator are NOT yet bugfree. The state of the compiler and simulator make it difficult to test code, since the programmer has no way of knowing whether he/she made a mistake or whether the error is caused by the compiler/simulator.

Solution   Serialize the the user's XMTC program so that it can be compiled using a stable, serial compiler (such as g++ or Microsoft Visual C++).

6.1.3 Assumptions and Restrictions

The following is a (potentially incomplete) list of known restrictions:

1. Do not use `typedef`s and `enum`s; they are not currently no supported in XMTC.

2. Recursive `includes` (including a header file inside a header file) are not supported.

3. All header files should be in the same directory as the source C file.

4. `ps`, `psm`, `spawn` and `sspawn` functions should not be used in define directives of C preprocessor. Serializer will not convert those functions to serial versions if they are used in that manner.

6.1.4   XMTC Serializer

The Serializer generates a copy of your code that can be compiler with traditional compilers. GCC and Microsoft Visual C were both tested during development.

Serializer consists of two executables: serXMTC and memReader. The latter is not directly called by the user however it has to be in a directory that is on the system path (preferably the same directory as serXMTC executable).

6.1.5   Walkthrough

1. Assume there exists a program 'mycode.c'.

   To Serialize 'mycode.c' for gcc the user executes

   `./serXMTC mycode.c`

   To Serialize 'mycode.c' for a Microsoft Compiler, the user executes:

   `./serXMTC mycode.c -win32`

2. In addition to 'mycode.c', assume 'xmtData.bin' and 'myMemHeader.h' files were generated by the memory map tool and 'myMemHeader.h' is included in 'mycode.c'.

   To Serialize 'mycode.c' with memory map for gcc the user executes

   `./serXMTC mycode.c -memload myMemHeader`

   To Serialize 'mycode.c' with memory map for a Microsoft Compiler, the user executes:

   `./serXMTC mycode.c -win32 -memload myMemHeader`

3.
   - The Serializer will generate files called `serializing_header.h` and `serial_mycode.c`. For all the header files included in `mycode.c` it will also generate `serial_headerfile.h`.

   - The Serializer will parse `mycode.c`, recursively calling itself on any locally included file (`#include "filename"`). Note that system libraries are not supported in XMTC.

   - FOR EVERY FILE!
     The Serializer will replace parallel keywords and constructs in EVERY file that it processes:

     ```
     * int or long...............XMTINT64
     * register int..............XMTINT64
     * psBaseReg.................XMTINT64
     * spawn(low, high).........for (int _I=low, _I<=high; _I++)
     * sspawn(newID)............high++; newID = high;
     * $........................._I
     * ps(LR, GR)...............temp = GR; GR+=LR; LR = temp;
     * psm(LR, GR)..............temp = GR; GR+=LR; LR = temp;
     * printf("%d").............printf("%lld") or printf("%I64d")
     ```

- The Serializer will insert the following line at the top of 'serial_mycode.c':

  `#include "serializing_header.h"`

- The Serializer will insert the initial values for all the global variables defined in the serial version of the memory header file.

4. At this point the user may compile the generated files using a C compiler:

`gcc serial_mycode.c`

Include directives in the source C file are automatically converted to the new header filenames therefore user does not need to make manual changes.

Should any errors occur during compilation, the user should examine the generated files for obvious errors. For assistance, contact the author of XMTC Serializer (See the Contacts section).

Should any errors occur during compilation, the user should examine the generated files for obvious errors. For assistance, contact the author of XMTC Serializer (See the Contacts section).

6.1.6 Options

Running `./serXMTC -h` will print the following help:

```
-------------------------------------------------------
Usage:
    serXMTC <filename> [options]

Options:
   -win32: Serialize for Microsoft Visual C

   -memload <filename>:
           Read files generated by the memory map tool.
           <filename>.h and xmtData.bin will be read from the
           current directory

   -h:     Show this help
-------------------------------------------------------
```

This section disscusses the options in more detail.

1. -win32

   When the '-win32' options is used, the Serializer produces code that can be compiled with Microsoft Visual C++. If the flag is not passed, the Serializer will produce code for g++.

   The key differences between the Visual C++ and g++:

   - Visual C++ uses `__int64` for 64-bit integers.
     vs.
     g++ uses `long long`

32

- Visual C++ uses `printf("%I64d", 64-bit int)`

    vs.

    g++ uses `printf("%lld", 64-bit int)`

2. -memload <base_name> If the mem load option is used the memory map will be read from files <base_name>.txt and <base_name>.h. These files should be created using the memory map tool and should not be edited manually. serial_<base_name>.h file will be created as the output.

3. -h

    When the '-h' option is used, the Serializer prints the help screen and exits without processing any files.

6.1.7 Contact for Support in XMTC Serializer

Please contact XMT Research Team

Chapter 7

Compiling and Simulating an XMTC Program

7.1 Overview

The overview of the XMT toolchain and the flowchart for compiling and simulating a program is shown in Figure 7.1. The preparation of the external data set for simulation (marked **1** in Figure 7.1) is covered in Chapter 5. This section explains the parts from **2a** to **5** shown in Figure 7.1.

Commands for the Compiler and Simulator   The main command for invoking the compiler is `xmtc`, and the main command for invoking the simulator is `xmtsim`. This chapter will explain the usage of these tools with various options.

Warning: Simulation Modes   The current XMT simulator has two simulation modes:

- **Assembly Simulation** This mode simulates all threads sequentially starting with the thread with smallest ID. This mode is faster. However, it represents a particular ordering of threads, and if the code is not written carefully, the output may not match the output of a real execution, where thread instructions are interleaved and executed concurrently.

- **Cycle-accurate Simulation** This mode simulates all threads cocurrently by interleaving the execution of instructions. This is the closest mode of simulation to a real processor. Due to the more complicated simulation process, the simulation time is significantly longer.

The simulator runs in *assembly simulation* mode by default. For the *cycle-accurate simulation* mode, the user needs to use the `-cycle` option. The simulator options are explained in detail in Section 7.4.

7.2 Inclusion of Memory Map

Currently, the XMT Simulator loads the external data into a portion of the memory that is normally used by global variables in C-language. In order to relate the actual data to variables that are used in the XMTC code, the *Memory Map-Header File* is used. The generation of this file is explained in Section 5.
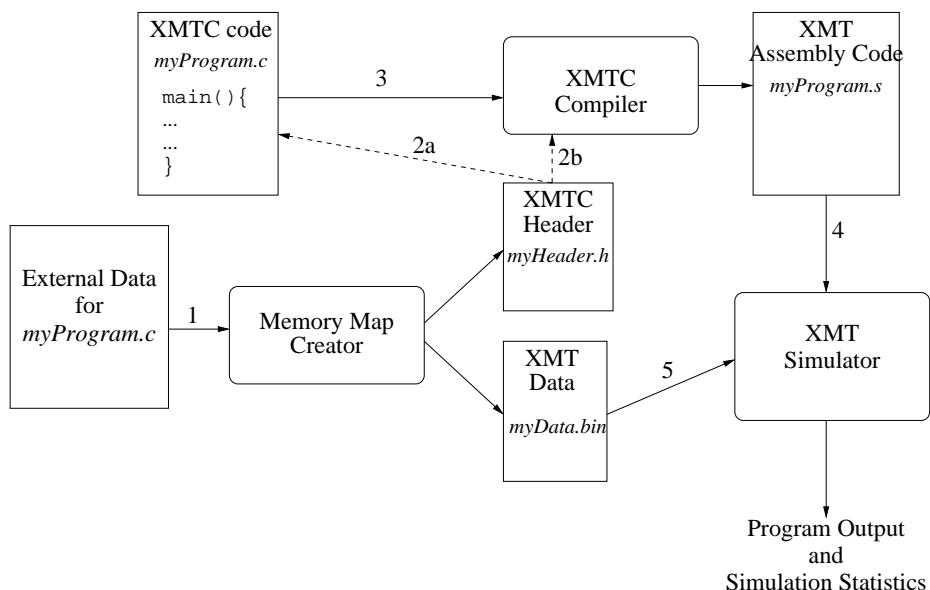
Figure 7.1: XMT Toolchain Overview: **1**. Preparing the external data set for simulation, **2a**. Inclusion of the header file by `#include` directive, **2b**. Inclusion of the header file by `-include` compiler option, **3**. Compilation of the XMTC code (`xmtc` command) **4**. Simulation of the XMT Assembly code (`xmtsim` command) **5**. Loading the XMT Data to the simulator by `-binload` option

There are two methods of using this file during compilation, which are shown as **2a** and **2b** in Figure 7.1:

2a. Include the file in the XMTC code using the `#include` directive

2b. Include the file by command line option `-include` during compilation

Both methods are identical in terms of outcome. The options ensure compatibility with the actual gcc compiler, which provides the same options to the user for inclusion of header files.

Suppose that, as in Figure 7.1, the XMTC code file is called `myProgram.c`, and the header file is called `myHeader.c`. To include this header file using the first method (**2a**), the following line has to be added to the top of `myProgram.c`:

```
#include "myHeader.h"
```

To include this header file using the second method (**2b**), the following command can be used at the system prompt:

```
xmtc myProgram.c -include myHeader.h
```

7.3 Compilation of the XMTC Code

The compilation of the XMTC code is marked with **3** in Figure 7.1. This section describes the common command line options for the XMTC Compiler (`xmtc`). The developer options are not described in the current version of this document. We will include them in the future, as their usage

becomes sufficiently mature. To see the full list of options, use the `xmtc -h` command. The list of the options is current as of version **0.9** by **Feb-13, 2006**.

Usage:

```
xmtc [-include <header.h>] [-D <MACRO>[=<VALUE>]] [-dirty] [-quiet] <file.c>

xmtc -clean

xmtc -h|-help

xmtc -version
```

-clean This option removes the non-essential files generated with **-dirty** option.

-D <MACRO>=<VALUE> Sets the value of C-Macro `MACRO` to `VALUE`. The effect is the same is if the directive `#define MACRO VALUE` is written in the first line of the XMTC code. This option is similar to the `-D` option of gcc compiler. If the `VALUE` is omitted, the value of the `MACRO` will be *True*, similar to the effect of the directive `#define MACRO`.

-dirty Some intermediate files are generated during compilation. These files are removed by default, after the compilation is finished. If the compiler is invoked with this option, the intermediate files will not be removed.

-h or -help Displays the on-line help message

-include <header.h> Includes the `header.h` file during compilaton. This is explained in detail in Section 7.2.

-quiet Some information on different compiler stages is shown during compilation by default. This option turns off these messages.

-version Displays the version numbers for each component of the XMTC Compiler tool.

The input to the compiler is the XMTC file with the extension `.c`. The output of the compiler is a human-readable assembly file with the extension `.s`. For example, if the XMTC file is called `myProgram.c`, then the output of the simulator is an XMT assembly file called `myProgram.s`. This file will be used as an input to the simulator.

7.4   Simulation of XMT Assembly Code

The simulation of the XMTC assembly code is marked with **4** in Figure 7.1. This chapter describes the common command line options for the XMT Assembly Simulator (`xmtsim`). The developer options are not described in detail in this version of this document. We will include them in the future, as their usage becomes sufficiently mature. To see the full list of options, use the `xmtsim -h` command. The list of the options is current as of version **0.9** by **Feb-13, 2006**.

Usage:

```
xmtsim [-binload|-textload <file>]
       [-bindump|-textdump <file>]
       [-out <file>]
       [-cycle|-count]
       [-trace]
       <file.s>

xmtsim -check

xmtsim -h|-help

xmtsim -version
```

-bindump <file> This option dumps some part of the XMT memory in binary format to `file` after the simulation is finished. The portion that is dumped is the same portion that is loaded to the simulator through `-binload` or `-textload` options. The format of the generated file is the same as the *Memory Map–Binary File* that is generated by the *Memory Tool* as described in Section 5.3.

-binload <file> This option is used to load external data into the simulator. It will be explained in detail in Section 7.5.

-check This option runs a short self-test routine.

-count This option reports the number of executed instructions. In addition to the total instruction count, the numbers of specific instruction groups are also reported. This option cannot be used together with the `-cycle` option.

-cycle This option reports the number of XMT clock cycles that are executed during the simulation. This option cannot be used together with the `-count` option.

-h or -help Displays the on-line help message

-out <file> During simulation two types of messages will be displayed on the screen: The messages from the XMTC code, which are generated by the printf statements (see Section 4.3.2), and the informative messages from the simulator including warnings and errors. If this option is used, all of these messages will be written in `file` instead of being displayed on the screen.

-textdump <file> This option dumps some part of the XMT memory in text format to `file` after the simulation is finished. The portion that is dumped is the same portion that is loaded to the simulator through `-binload` or `-textload` options. The format of the generated file is the same as the *Memory Map–Text File* that is generated by the *Memory Tool* as described in Section 5.3.

-textload <file> This option is used to load external data into the simulator. It will be explained in detail in Section 7.5.

-trace If this option is used the simulator displays the dynamic instruction trace while the simulation is running. After each assembly instruction is executed the contents of the relevant registers is displayed as well.

-version Displays the version numbers for each component of the XMT Assembly Simulator tool.

7.5   Loading Data into XMT Simulator

Loading external data into the XMT simulator is marked as **5** in Figure 7.1. We will first describe how to load using binary files, as shown in Figure 7.1. Later, we will describe an alternative method of loading data into the simulator using text files. Finally, we will discuss the advantages and disadvantages of both methods.

7.5.1   Binary Data Files

There are mainly three methods to generate a binary data file for XMT simulation:

1. Use the *Memory Tool* and generate a *Memory Map–Binary File*. Section 5.3 explains the *Memory Tool* and how to generate a proper binary data file that can be loaded into the simulator. This method should be used for external data sets.

2. Use the `-bindump` option of the XMT assembly simulator, to generate a *Memory Map–Binary File* that represents the memory contents after the simulation is finished. In some cases, a user may want to use the output of an XMTC program as input to another XMTC program with exactly the same memory mapping. In this case this second method can be used to carry the output of the first program to the second one.

3. Create a binary file using some external tool. We do not recommend nor support creating a binary file using an external tool.

Suppose that the *Memory Map–Binary File* that you want to load into simulator is called `myData.bin`, and the XMT assembly file is called `myProgram.s`, as shown in Figure 7.1. Then, the command to simulate the assembly file using that binary file is:

```
xmtsim -binload myData.bin myProgram.s
```

7.5.2   Text Data Files

There are mainly three methods to generate a text data file for XMT simulation:

1. Use the *Memory Tool* and generate a *Memory Map–Text File*. Section 5.3 explains the *Memory Tool* and how to generate a proper text data file that can be loaded into the simulator. This method should be used for external data sets.

2. Use the `-textdump` option of the XMT assembly simulator, to generate a *Memory Map–Text File* that represents the memory contents after the simulation is finished. In some cases, a user may want to use the output of an XMTC program as input to another XMTC program with exactly the same memory mapping. In this case this second method can be used to carry the output of the first program to the second one.

3. Create a text file using some external tool such as a text editor. The user must be careful to match every variable declared in the *Memory Map–Header File* to one proper value in this text file. The values have to be separated by whitespace characters (such as space, tab or new line). Alternatively, the users can modify the values in a text file that is generated by one of the earlier methods.

Suppose that the *Memory Map–Text File* that you want to load into simulator is called `myData.txt`, and the XMT assembly file is called `myProgram.s`. Then, the command to simulate the assembly file using that binary file is:

```
xmtsim -textload myData.txt myProgram.s
```

7.5.3   Comparison of Data Loading Methods

Advantages of working with **binary** data files:

1. In general, binary representation of 64-bit data takes less disk space than text representation.

2. If some data is represented in text, some precision might be lost during translation, depending on the text read/write functions that are used. Binary format does not lose precision, because all 64 bits of the value are read or written as intended.

 Advantages of working with **text** data files:

1. In specific cases the text representation uses less space. For example, the text representation of integer 0 with the separating whitespace takes two characters (two bytes), whereas its binary representation takes 8-bytes.

2. The text representation is easily readable and modifiable. Therefore it might be preferred during development stage of a program, where the user might want to quickly modify some data values for debugging purposes.

Part IV

Warning and Error Messages

Chapter 8

Error Messages

8.1 Simulator Error Messages

This chapter lists the error and warning messages that a user can receive from the XMT Simulator. The messages are divided in context-based sections, and unless mentioned otherwise, they are ordered alphabetically in each section. Some messages that are relevant in multiple context are duplicated for easier access. The messages are listed as the message text first and some description following that. Detailed description is omitted in some messages that are self-explanatory (such as *Option should be first argument*).

8.1.1 General Errors

```
Please contact the author of this software with your source code and this
error message:
Class:  name Class version:  version
supplementary info
```

> Please send the error text (including the class name, version and supplementary information), all related files (such as code and data files) and the output of the `xmtsim -version` command to the appropriate contact person.

8.1.2 File Input/Output Errors

```
Cannot create memory dump file.
```

```
Input file could not be read:  filename
```
> The mentioned file is not readable. Check the file names and permissions

```
Help file not found.
```
> This exception might occur as a result of -h or -help option if for some reason the help text could not be read.

```
Help file could not be read.
```

This exception might occur as a result of -h or -help option if for some reason the help text could not be read.

`Output file could not be written:` *filename*

The mentioned file is not writable. Check the file names and permissions

8.1.3 Errors Related to Use of Arguments

Following error messages are for incorrect use of arguments. Most of the messages are self-explanatory. For correct usage, type: `xmtsim -h` or `xmtsim -help`. The messages in this section are listed in alphabetical order based on the relevant simulator option.

`Option not known:` *string*

This message is displayed when there is an unknown command line parameter starting with a "-" character.

`Wrong number of arguments.`

This message is displayed when there is an unknown command line parameter that is not starting with a "-" character.

bindump

`Cannot create memory dump file.`

`Option requires argument:  -bindump <filename>`

`Wrong use of option:  -bindump`

This message is displayed when there already is another -textdump/-bindump option.

binload

`Option requires argument:  -binload <filename>`

`Wrong use of option:  -binload`

This message is displayed when there already is another -textload/-binload option.

out

`Option requires argument:  -out <filename>`

`Output file could not be created:` *filename*

This message is displayed when the out file could not be created.

textdump

```
Cannot create memory dump file.
```

```
Option requires argument:  -textdump <filename>
```

```
Wrong use of option:  -textdump
```
This message is displayed when there already is another -textdump/-bindump option.

textload

```
Option requires argument:  -textload <filename>
```

```
Wrong use of option:  -textload
```
This message is displayed when there already is another -textload/-binload option.

8.1.4   Runtime Errors

`Local register value cannot be value for ps (register).`
This happens if a the local register in the prefix sum has a value greater than 1.

`Trying to read uninitialized memory location:  address`
This message is displayed if -checkmemreads or -warnmemreads option is used. Depending on which option is used it will be an error (checkmemreads) or warning (warnmemreads). It indicates that a memory location is accessed unexpectedly, and it is the equivalent of (actually more powerful than) segmentation fault.

8.1.5   Errors Related to Assembly File Parsing

These messages are not likely to occur if you compile your program from XMTC source, and do not modify the resulting assembly file manually. Some of the messages also report the location of the error by line and column number. In general, these errors are caused by some portion of the code that cannot be parsed correctly.

`Cannot use pseudo-instructions in simulator:  instruction`

`data file name contains a non-integer value:  value`
Well this is one parse error that compiler has nothing to do with.

`expecting ...`
Parsing error from underlying compiler. Please contact XMT Team

`Label string could not be found.`
This happens if the target of a jump instruction with label could not be found. Might happen if a function is declared but not defined.

`(Un)signed value ` *`number`* ` cannot be expressed with ` *`number`* ` bits.`
>   This happens if an immediate value is outside the bit bounds of a fields.

8.1.6   Errors Related to Assembly File Simulation

These messages are not likely to occur if you compile your program from XMTC source, and do not modify the resulting assembly file manually. Unlike the previous section, these messages are in general caused by logical errors in programming (such as division by 0), or use of some syntactically correct instruction within an inappropriate context (such as a spawn instruction in a spawn block).

`Cannot use 0 register as destination.  This is a read-only register.`

`Divide by 0.`
>   Thrown by div and divu.

`JOIN instruction is not allowed in Master TCU.`

`Master TCU cannot have broadcast instruction.`

`Overflow in addition of ` *`reg`* ` and ` *`reg`* `.`
>   Thrown by add, addi, sub and subi.

`Reached end of file without a HALT instruction.`

`register name` ` is not a register of ` *`tcu name`* `.`
>   This will happen if a TCU tries to access a global register by means other than ps (or a register number is greater than the register file size).

`Shift amount is out of bounds:  ` *`value`*
>   Happens with shift type instructions where shift value is stored in a register.

`SPAWN instruction is not allowed in TCUs.`

`Trying to execute ascii directive:  ` *`text`*
>   If the assembly file reaches an ascii (printf) directive on the execution path.

`Trying to print a non-ascii block:  ` *`text`*
>   If the target of a prn or a prnln instruction is not an ascii directive (text), this exception will be thrown.

44

Part V

Glossary

Chapter 9

Legend and Index

9.1 Legend

In this manual, we used:

- *italic characters* for XMTC-related terms (e.g. *Master TCU*, *Parallel Section*).

- `fixed-width characters` for XMTC code, reserved names, file names and shell commands (e.g. `#include`, `spawn`, `xmtsim -binload xmtData.bin xmtAssembly.s`).

9.2 Index

This section defines, describes and provides pointers to some key terms mentioned in this manual.

Content File is a text file prepared by the programmer. It contains the initial values of an array of integer or double FP numbers. As the array variable is created using the *Memory Tool*, this file can be used to initialize the values within the array. Section 5.3.4.

Initialization Block is enclosed within curly braces({...}) after every `sspawn` statement. This block is executed before the newly spawned *virtual thread* starts its execution. Section 2.2

Master TCU is responsible of executing all of the *Serial Section* within the XMTC code. *Master TCU* is inactive during parallel execution. The XMT architecture envisions a contemporary superscalar processor with MIPS ISA for the *Master TCU*.

Memory Map - binary file is a file containing a part of the memory that is fed into the simulator. This file is used for providing external input to an XMTC program.

Memory Map - header file is the file to be included (using `#include`) in the XMTC source code. This allows proper compilation and execution with external inputs and memory allocation.

Memory Map - text file has the identical contents as the *Memory Map - binary file* but in text format. It can be used for the programmer's reference and debugging purposes. This file is also required by the serializer.

Parallel Mode is the execution mode of the computer, where the *TCU*s execute multiple *virtual thread*s at once.

46

Parallel Section is a part of the code that is assigned to *virtual thread*s, and executed by *TCU*s in parallel. Also, see *spawn* and *Spawn block*.

ps Special XMTC Statement. Computes **p**refix **s**um using a *psBaseReg* type variable as a base. Section 2.3.

psBaseReg is a type identifier for variables that are stored in a small set of architectural registers. These variables can be accessed (read or written) regularly by the *Master TCU* within the *Serial Section*. Within the *Parallel Section* they can only be accessed using **ps** statement. These variables are declared **globally** as `psBaseReg`.

psm Special XMTC Statement. Computes **p**refix **s**um to **m**emory using any regular integer variable as a base. Section 2.3

Serial Mode is the execution mode of the computer, where the *Master TCU* executes one of the *Serial Section*s in the code.

Serial Section is a part of the code that is executed serially by the *Master TCU*

Serialization is the act of converting the XMTC program into a regular C program, by executing all *Virtual Thread*s in the interval [*begin, end*] sequentially. Currently this is the only method for ensuring functional correctness of an XMTC program.

spawn Special XMTC Statement. It defines a part of the code (*Spawn Block*) that is executed by several (all) available *TCU*s in parallel. The execution is in SPMD (Single Program Multiple Data) principle. Section 2.1.

Spawn Block is the part of the code that is executed in SPMD fashion. It is enclosed within curly braces ({...}) after the `spawn` statement. This block defines the functionality of a *Virtual Thread*. Section 2.1.

sspawn Special XMTC Statement. It stands for "single spawn". This statement can be used to spawn more *Virtual Thread*s (one at a time) from within the *Spawn Block*. `sspawn` is followed by an *Initialization Block* that is executed by the current thread. The newly spawned *virtual thread* executes the same *Spawn Block* in which the `sspawn` resides. Section 2.2.

Thread Control Units (TCUs) are the processing units that execute the statements of the *Virtual Thread*s within the *Spawn Block*.

Thread ID is the means for distinguishing *Virtual Thread*s from each other. Every *Virtual Thread* has a unique *Thread ID*. This number can be accessed using `$` character within a *Spawn Block*. The *Thread ID*s are assigned sequentially, starting with the value of the first parameter of the `spawn` statement Sectr 2.1.

Virtual Thread is a basic work unit in XMTC. There is no limit on the number of *Virtual Thread*s. During parallel execution, each available *TCU* executes a virtual thread. The code to be executed as *Virtual Thread*s is defined in *Spawn Block*.