Performance of the Ultra-Wide Word Model

by

Camila Pérez Gavilán Torres

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2017

Abstract

The Ultra-wide word model of computation (UWRAM) is an extension of the Word-RAM model which has an ALU that can operate on w^2 bits at a time, where w is the size in bits of a cell in memory. The purpose of this thesis is to explore the applicability of the UWRAM model, particularly when compared to the PRAM model, from an algorithmic point of view, to determine its potential for common applications.

The work is divided into three sections: First we describe the model, its instruction set, strengths and weaknesses, and provide a few small examples that showcase the functionality of the model and how simple techniques can be used to speed up sequential algorithms. In the second section, we discuss the problem of sorting and searching, and show that elaborate data structures such as the fusion tree can be easily adapted to the model, allowing the sorting of n integers in $O(n\frac{\log n}{\log\log n})$ time with small constant factors. Lastly, we provide simulations of UWRAM and PRAM programs to solve two problems: subset sum and string matching. In the first case we show how a dynamic programming algorithm can be sped up using bit parallelism where traditional parallelism is difficult to achieve, and in the second, we show that even in a problem that is simple to parallelize traditionally, the UWRAM can perform well when compared to a PRAM.

Acknowledgements

First of all, I would like to thank Professor Alex Lopez-Ortiz, who was always an inspiration with his extensive knowledge, his bright research ideas, his stories and his positivity, even through difficult times.

I would like to thank Professor Ian Munro for his support and advice, his insightful questions, clever ideas and broad knowledge, without which the completion of this work would not have been possible.

I would also like to thank my friends from the algorithms group, particularly Saeed, Amit, Hisham and Martin, for their friendship, advice, and great conversation. Finally I would like to thank all the great friends I've made during my stay in Canada, and my family, who always had a word of support or advice in good and bad times.

Dedication

To my parents, who always encourage me, and never let me doubt myself.

Table of Contents

Chapter 1

Introduction

Ever since the creation of computers and processors, great research efforts have been made to improve their performance. For many years, improvement was achieved through miniaturization, decreasing the size of transistor gate length and thus the clock rate of the processor. As transistor size approaches its physical limits, however, efforts have shifted toward exploiting parallelism to increase computational power [33, 16]. For example, Intel has not upgraded the transistor gate length in its popular desktop Intel Core series since 2008, but they have more than doubled the number of cores in them. Older versions had up to four cores and recent releases have up to ten, while gate length has remained between 14nm and 32nm throughout, depending on the version [36].

Parallelism can be achieved in several different ways, and is usually grouped by the type of instruction and data streams that the processor can manage into the categories proposed by Michael J. Flynn in [12]. This categorization is commonly known as Flynn's taxonomy.

Flynn classifies computers as either SISD, SIMD, MISD or MIMD. SISD computers (Single Instruction Single Data) are traditional sequential computers. Models of computation such as the RAM and Word-RAM fit into this category. Vector or array computers fit into the SIMD (Single Instruction Multiple Data) category; they were devised to solve problems where relatively simple tasks need to be executed over massive amounts of data. Typical applications are scientific calculations and computer graphics. Many Intel processors come equipped with vector registers of 128 to 256 bits, into which smaller types can be packed and operated on simultaneously through special instruction sets [21]. Current graphics processing units (GPUs) are also members of this group. The MISD (Multiple Instruction Single Data) category is mostly theoretical and has no well-known applications. However, specialized hardware could make use of an architecture such as this one; in signal processing for example, a MISD computer would allow multiple frequency filters to be applied to a single signal simultaneously [2]. The MIMD (Multiple Instruction Multiple Data) category is the most explored, and includes all multiprocessors and multi-core architectures.

The MIMD category is so popular in fact, that it can be further categorized depending on the computer's memory structure and synchronization techniques [24]. Johnson divides MIMD computers into GMSV (Global Memory Shared Variables), GMMP (Global Memory Message Passing), DMSV (Distributed Memory Shared Variables), and DMMP (Distributed Memory Message Passing). A PRAM, for example, would be considered a MIMD GMSV model.

In this thesis we will explore a particular type of SIMD parallelism known as bit parallelism. Bit parallelism exploits the fact that in current architectures with larger word sizes (32 or 64 bits), many types are smaller than the processor word and so several elements can be packed and operated on simultaneously. In a 64 bit architecture, eight 8-bit characters can be operated on at once. It is a curious type of parallelism, since it can be executed in a purely sequential computer.

The theoretical model which allows, and in fact, was defined to analyze bit parallelism is called a Word RAM, and was defined by Torben Hagerup in 1998 [17]. Although there is plenty of research improving algorithms by using bit parallelism [15, 6, 17], the speedups achieved are small due to limited word sizes.

In 2014, Farzan, Lopez-Ortiz, Nicholson and Salinger proposed a model specifically to exploit bit level parallelism, which they called the Ultra Wide Word Model (UWRAM) [11]. They introduce a processor with word size ranging from 1,000 to 10,000 bits, while maintaining regular word size in memory. There are two main differences between their model and vector processors, which at first glance seem to be the most similar. Firstly, memory access operations in the UWRAM are a lot more flexible than in vector processors, which are limited to consecutive or evenly spaced access. The UWRAM allows arbitrary access to memory, and is in that aspect a lot closer to the PRAM model. The second important difference is that the UWRAM does not have any boundaries dividing the large word, increasing again the flexibility of the model. A full description of the model is given in chapter 2.

The goal of this thesis is to explore the UWRAM model in greater depth and to analyze its applicability from an algorithmic point of view in order to determine its potential for common applications. In chapter 2 we will review the details of the model, introduce some new instructions to improve its performance, and discuss some small examples that show the general feel of the model. In chapter 3 we will discuss the problem of sorting and searching in this architecture, and in chapter 4 we will present simulations of two problems, one which is considered traditionally easy to parallelize (String Matching) and one which is not (Subset Sum), and compare them to the performance of a PRAM (essentially a stronger parallel model), showing in one case that the UWRAM can perform reasonably well against a PRAM, and can improve the worst-case run-time of the algorithm, and in the other, that the UWRAM can significantly outperform, thus showcasing the type of techniques that can be used to devise efficient algorithms for the UWRAM model, and the advantages that can be obtained with the use of flexible large scale bit parallelism.

Chapter 2

The Ultra-Wide Word Model

The Ultra-Wide Word-RAM model (UWRAM), proposed by Farzan, et al. in their 2014 paper [11], is an extension of the Word-RAM model, or simply Word model, as described by Hagerup [17]. In the Word-RAM model, each memory location is w bits long and stores an integer in the range $\{0, ..., 2^w - 1\}$. The instruction set includes basic arithmetic operations (addition and subtraction), binary operations (AND,OR and NOT), conditional and unconditional jumps, right and left shifts, and store and load operations. We will try to keep this model as close to the restricted model as possible, so multiplication and division will not be supported. Each operation, including memory access, is assumed to take unit time.

The UWRAM introduces an ALU which works on w^2 bit ultra-wide words, from here on referred to as ultrawords, which are divided into w bit blocks for addressing purposes, but are treated like a single word by the ALU; this means that basic Word-RAM operations can be used on entire ultra-words at once. The model maintains a w bit ALU additional to the ultra-wide one , and maintains w bit memory addressing.

Before we continue with the description of the model, it is important to define the notation we will use. This notation is very straightforward and can be found in Table 2.1.

Since the UWRAM allows parallel access to memory, it is necessary to define its memory access type. The model will accept concurrent reads from a memory location, but not concurrent writes. This access type is known as CREW (Concurrent Read Exclusive Write). If a programmer should attempt to write to the same memory location from more than one block of an ultraword at a time, no assumptions can be made as to the result written in the cell, since these assumptions lead to stronger models that are sub-versions of the CRCW (Concurrent Read Concurrent Write) access types; for example, CRCW-R will

W_k	ultraword k
$W_k[i]$	block i of ultraword k
$W_k[i][j]$	j^{th} bit of block i of ultraword k
w_k	word k
$w_k[i]$	bit i of word k
w	word size in bits
u	ultraword size in bits

Table 2.1: UWRAM notation

write a random word into memory from the ones attempting to be saved, and CRCW-S, CRCW-A and CRCW-X will store reductions of the words using sum, logical and or logical xor respectively [29]. For this reason, the programmer must assume that anything stored in the conflicting cell will be completely useless.

The original definition of the model introduced three types of memory access operations, which allow access by block, ultra-word or content for both reads and writes, as can be seen in the following instructions:

#Read access by block
uwlb $ud j base *# U_d[j] = MEM[base+j]*

#Read access by ultra−word
uwluw $ud base *# for every j in U_d: U_d[j] = MEM[base+j]*

#Read access by content
uwlc $ud base *# for every j in U_d: U_d[j] = MEM[base+U_d[j]]*

Write access follows the same patterns, though in the write access by content, the programmer must be careful not to attempt to access the same location more than once.

#Write access by block
uwsb $ud j base *# MEM[base+j] = U_d[j]*

#Write access by ultra−word
uwsuw $ud base *# for every j in U_d: MEM[base+j] = U_d[j]*

#Write access by content
uwsc $u1 $u2 base *# for every j in U_1: MEM[base+U_2[j]] = U_1[j]*

Through these instructions, we make the assumption that the model can access any w cells in memory in parallel and in constant time (Figure 2.1). This memory model is completely theoretical; in practice, memory access would need to be implemented through an interconnection network, and due to the extensive number of memory slots, blocks of slots would need to share network access points [29]. It is important to note that this assumption is usually made by popular shared memory multi-processor models such as the PRAM, which is in a way the closest model to the UWRAM, and the one we will be comparing it to. Latency produced by memory accesses can also be mitigated in practice by caching [11], so the assumption is not unreasonable. A more detailed description of the PRAM can be found in section 2.2. Since both models we will compare suffer from the same memory access limitations, this assumption will not affect results.

A physical implementation of a UWRAM would require a large data bus to handle concurrent access to memory. A UWRAM with $w = 32$ would require a data bus of at least 1024 bits to access w words concurrently, and one with $w = 64$ would need 4096. There are GPUs in the market today with data buses of those sizes, showing that the requirement is not unreasonable. For example, Nvidia's Quadro family GPUs have data bus sizes ranging from 64 to 4096 bits [27].

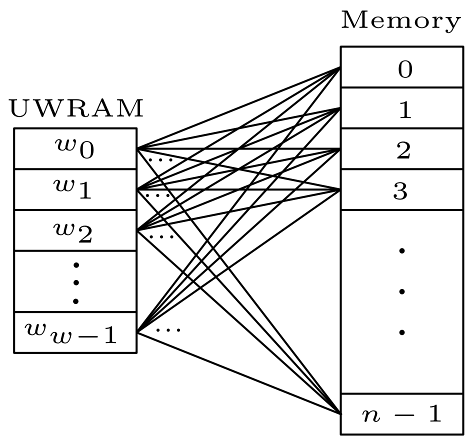

Figure 2.1: Conceptual network required for concurrent memory accesses assumed by the model [29]

2.1 Additional Instructions

Since the main motivation behind the UWRAM model is to be able to operate on w elements of w bits at once, it makes sense to provide the model with instructions to aid in this task. The following instructions facilitate memory access for regularly spaced words and arithmetic operations. Even though the model allows operations for any field size $f \leq w^2$, field size w is the most natural, and should be simple to deal with.

We will maintain the UWRAM instructions introduced in [11]: *compress* and *spread*. The *compress* instruction packs the first bit of each block of an ultraword into the first word of another one, such that if $W_2 = compress(W_1)$, then $W_2[0][i] = W_1[i][0]$ for $0 \leq i < w$, and the rest of the blocks are set to zeros. The *spread* operation is the opposite of compress, and spreads each bit of the first word of an ultraword into the first bit of each block of another one, such that if $W_2 = spread(W_1)$, then $W_2[i][0] = W_1[0][i]$, and all other bits are set to zeros.

$W_1 =$	$1\ X\ X\ X$	$0\ X\ X\ X$	$1\ X\ X\ X$	$1\ X\ X\ X$
$compress(W_1) =$	$1\ 0\ 1\ 1$	$0\ 0\ 0\ 0$	$0\ 0\ 0\ 0$	$0\ 0\ 0\ 0$

$W_1 =$	$1\ 0\ 1\ 0$	$X\ X\ X\ X$	$X\ X\ X\ X$	$X\ X\ X\ X$
$spread(W_1) =$	$1\ 0\ 0\ 0$	$0\ 0\ 0\ 0$	$1\ 0\ 0\ 0$	$0\ 0\ 0\ 0$

Figure 2.2: Examples of the *compress* and *spread* operations, with $w = 4$. Bit values that are not relevant for the final result are marked with an X.

We propose the addition of three more ultraword instructions: *replicate*, *spread_sequence* and *uwpopcnt*. The instruction *replicate* arises from the need to operate over every block of an ultraword with a single w sized value. Simple operations like adding a constant to every element of an array, or applying a bitmask to a set of elements, or even comparisons to zero require replicating a w sized value into each block of an ultraword. Without an instruction to enable this operation, it requires the execution of $log(w)$ instructions. The instruction *replicate* can then be defined the following way: if $W_1 = replicate(w_0)$, where w_0 can be any w sized word, then $W[i] = w_0$ for $0 \leq i < w$. In a word RAM, this effect can be achieved through multiplication, which our model does not allow. Replication can also be achieved by reading from the same cell simultaneously using a read by content instruction, but this in turn would require the address of the cell to be replicated into an ultraword, so it is logical to have an instruction specifically for this purpose. Replication is most natural with field size $f = w$, but can be easily implemented with additional shifts and ors for $f < w$, as long as f is a power of two.

We add the instruction *spread_sequence* for the specific purpose of reading from and writing to evenly spaced blocks of memory, through the *read_content* or *write_content* operations, simply and in constant time. The power of these instructions, which are key to the functionality of the model, is greatly hindered if every address needs to be calculated sequentially. The instruction takes a single word sized parameter w_0 and outputs an ultraword holding the first w elements of an arithmetic progression with 0 as an initial term and a common difference of w_0. There are several situations where it is necessary to read in this spaced manner; for example, when reading from a large matrix stored in row or column major order, or when packing an ultraword with more than one element per word. This instruction is not in any way meant to cheat the "no multiplication" constraint of the model, and none of the examples we give use it for anything other than evenly spaced memory access. If the model were to be physically implemented, this instruction could be substituted with a memory access instruction taking a base and a stride as a parameter. We chose to keep this instruction for the simulations to avoid having too many types of memory access. Parallel access with a stride or spacing ≥ 1 is already a common instruction in processors today, since it is part of Intel's AVX instruction set [21], available in many Intel and AMD processors.

$w_0 =$	2			
$replicate(w_0) =$	2	2	2	2
$spread\_sequence(w_0) =$	0	2	4	6

Figure 2.3: Examples of the *replicate* and *spread_sequence* operations, with $w = 4$, in decimal representation.

The instruction *uwpopcnt* is an ultrawide version of the *popcnt* instruction, available in recent processors from Intel and AMD that support the SSE4.2 and ABM instruction sets respectively [20]. This instruction takes an unsigned integer word as input and outputs a count of set bits in that word (bits that are equal to 1). In the ultrawide version, instead of considering all bits in the ultraword, only the first bit in each block is considered. The instruction can be thought of as a *compress* instruction followed by a regular *popcnt*. Since the model does not support direct methods for field-size arithmetic, methods such as adding sentinel bits for subtraction must often be used. A combination of this technique and the *uwpopcnt* could be used, for example, to calculate the rank of an integer in an array of w integers in constant time, regardless of word size. Further use of this instruction will be explained in detail further on.

$$\begin{array}{r l} W_1 = & \begin{array}{|c|c|c|c|}\hline 1\ X\ X\ X & 0\ X\ X\ X & 1\ X\ X\ X & 1\ X\ X\ X \\ \hline \end{array} \\ uwpopcnt(W_1) = & \begin{array}{|c|}\hline \quad 3 \quad \\ \hline \end{array} \end{array}$$

Figure 2.4: Example of the *uwpopcnt* instruction, with $w = 4$.

2.2 Comparing with a PRAM

The PRAM is a generalization of the Word-RAM generally used to measure the complexity of parallel algorithms. The model is assumed to have an unbounded number of processors and unbounded shared memory cells, which are the only way the processors can communicate with each other [29]. Each processor also has local memory (registers) of its own, and has an additional register which stores its identifier. The instruction set for each processor is the same as that of a regular Word-RAM processor, and they operate the same way. The execution of any instruction in this model is also assumed to take unit time. The read and write memory accesses will be restricted by the CREW access type.

Another important characteristic of the model is the fact that all processors share a common clock. This means that all processors will execute an identical program, which they will execute in a synchronous manner, although clearly local memory and input may lead each of them towards a different execution path[18].

In order to activate processors for execution, there will be a single activation register, or slot in memory, to which the number of processors needed for the computation will be written. Processor $P0$ will have this responsibility, and will also halt all processors when it executes a halt instruction. Beside these extra responsibilities, $P0$ is a regular Word-RAM processor. Since the model allows concurrent reads, all models may read concurrently from the activation register. Processors with an identifier smaller than the number in the activation register will be activated and will immediately begin execution of the program.

In many cases, a reduction of each of the processor's results to a final unique result will be necessary. Since the model does not allow concurrent writes, this must be done through a tree like fan-in operation, which has logarithmic complexity.

In order to compare the models fairly and for the sake of simulation, the PRAM model cannot be unbounded in its number of processors. For this reason, we will use a small-PRAM, with w processors. This means that there will be w processors available, but a program may choose to use less. This restriction will give us w^2 bits to work with in each of the models.

In this section we will review a few small examples that will show the general feel of the UWRAM model and how it compares to the PRAM, making emphasis on examples where the UWRAM outperforms the PRAM. In these examples, we will refer to the number of processors (in case of the PRAM) or the number of blocks (for the UWRAM) as p, and to the word size in both models as w. We assume throughout the text that p is equal to w, but will differentiate to make algorithm analysis easier to understand.

2.2.1 Checking if every element in an array is zero

Given an array of integers $A = \{a_1, a_2, ..., a_n\}$, we output 1 if all elements are equal to zero, and 0 otherwise. The sequential algorithm for this problem is to simply loop through the array and to output a 0 if anything other than zero is found. This algorithm clearly has an $O(n)$ running time. To run this algorithm in the PRAM, n/p entries of the array are assigned to each processor, each processor checks each of these entries, and then a logarithmic fan-in operation is necessary to obtain the final result. This algorithm has a run-time of $O((nlog(n))/p)$, and thus a speedup to the sequential algorithm of $p/log(n)$. In the UWRAM, n/p entries are loaded into an ultra-word register, and can be compared to zero in a single operation resulting in a single value. Since the final result can be accumulated in a local register, no other operations are necessary and the run-time is $O(n/p)$, giving an optimal speedup of p over the sequential algorithm.

2.2.2 Transposing a bit matrix

Given an $n \times n$ matrix M of packed bits, stored in row-major order, find the transpose of M. This means one bit is one element, so a word in memory would store w elements of the matrix (Example from [35]).

$$\begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 8 & 9 & a & b & c & d & e & f \\ g & h & i & j & k & l & m & n \\ o & p & q & r & s & t & u & v \\ w & x & y & z & A & B & C & D \\ E & F & G & H & I & J & K & L \\ M & N & O & P & Q & R & S & T \\ U & V & W & X & Y & Z & \% & \# \end{bmatrix} \longrightarrow \begin{bmatrix} 0 & 8 & g & o & w & E & M & U \\ 1 & 9 & h & p & x & F & N & V \\ 2 & a & i & q & y & G & O & W \\ 3 & b & j & r & z & H & P & X \\ 4 & c & k & s & A & I & Q & Y \\ 5 & d & l & t & B & J & R & Z \\ 6 & e & m & u & C & K & S & \% \\ 7 & f & n & v & D & L & T & \# \end{bmatrix}$$

A sequential algorithm for this problem would require accessing each element of the matrix once, by reading the correct block from memory, masking out all but the required bit, and accumulating bits in a register until an entire word has been packed and can be saved to memory. Element $M[a][b]$ of the array can be found in the word $((a+1) \times (n/w)) - 1$ from the beginning of the array, in bit $b\%w$ (where bit 0 is the most significant). This algorithm is clearly $O(n^2)$.

A PRAM can access p memory blocks at once, in column order, out of which one bit will be taken to pack into a word and output as a block of a row. We see an example from the 8×8 matrix above, whith $w = 4$ where the first bit of each word is taken to form the first word of the first row of the transpose.

$$[0\ 1\ 2\ 3]\ \ [8\ 9\ a\ b]\ \ [g\ h\ i\ j]\ \ [o\ p\ q\ r]\ \ \longrightarrow\ \ [0\ 8\ g\ o]$$

This operation requires a logarithmic fan-in for the packing, and clearly still needs to visit every element of the matrix, which gives a run-time of $O(n^2 log(n^2)/p)$, and a speedup to the sequential algorithm of $p/log(n^2)$.

The UWRAM can perform this calculation in constant time, using the spread sequence operation for spaced memory access and the compress operation for bit extraction and packing. This gives a run-time of $O(n^2/p)$, and so a speedup of p.

2.2.3 Adding large numbers

Given two integers a and b with n bits each, find $c = a + b$. We assume that the number is stored in contiguous w sized blocks in memory. The sequential algorithm to solve this problem iterates through every block of the number, keeping a flag or register to indicate if there is a carry, so that it may be added to the next block. Since this algorithm must access each block once, it has a run-time of $O(n)$.

The PRAM splits two blocks of the large number to each of the processors, so that processor with index i will be responsible of adding $a[i] + b[i]$. Carries, if they exist, are stored into an array in memory, as described in [3], and then added to the next block in the following iteration until there are no more carries. In the worst case, there will be carries in every iteration, and so the run-time is of $O(n)$, gaining no speedup over the sequential algorithm.

The UWRAM can add two numbers of $w \times p$ bits in a single operation, and would only have to keep track of a carry bit for $n > w^2$. This results in a simple algorithm with a run-time of $O(n/p)$, and a speedup of p over the sequential.

2.3 Limitations

The UWRAM model is closely related to both PRAM models and VPUs in different ways. It is similar to vector processors in that a single instruction is executed over a large amount of data. Even though the UWRAM is more flexible in its field size and memory access, the VPU executes parallelism in a more straight forward way. Since field sizes are fixed, calculations of flags or carries are part of the instructions and are given at no extra cost. In the UWRAM, however, the programmer must handle possible overlap of fields and carries as the cost of greater flexibility. This will create overhead of instructions: when using sentinel bits for subtraction for example, it is necessary to split the subtraction operation into two parts, and then mask the result to access each field separately if needed.

The PRAM has the obvious advantage of being able to perform different operations on different data at the same time, which means it can execute different programs concurrently. The UWRAM cannot in any way compete with this, since it is still essentially a sequential model. In order to simulate a PRAM with a UWRAM, each different operation would have to be executed separately and on the right fields aided by bit masks. This would lead to a constant time overhead, where the constant would be a value related to the total number of instructions available in the PRAM [11]. The main strength of the UWRAM over the PRAM is that the different fields can interact with each other very easily through shifts or addition or subtraction operations. The "no boundary" model also allows decisions to be taken over all the fields at once without the need of fan-in operations.

2.3.1 Multiplication

An undeniable weakness of the model is the fact that it does not support multiplication. There are two reasons for this: one is the complexity of the multiplication operation, which depends not only on the number of bits in the numbers to be multiplied, but on their value. The other is the fact that the result of the multiplication of two n bit numbers may require as many as $2n$ bits to store the product. Assuming that multiplication would be carried out on two ultrawords without any concept of blocks or boundaries, as is the definition of the model for other arithmetic operations, this would not be possible, since the model does not support registers or memory accesses larger than w^2. This could be worked around by allowing the multiplication only of half-ultrawords, with $w^2/2$ bits, resulting in a full ultraword product. This leaves only the problem of the complexity of the operation, which is not a small hurdle to overcome.

There are many different algorithms to multiply two binary numbers, most of which can be implemented both in software and in hardware if enough resources are available. The Shift/Add algorithms are the simplest among these, and the most economical regarding space and hardware, but are also the slowest. These algorithms work essentially the way grade school multiplication works, with the additional observation that, when multiplying any bit of the multiplier by the entire multiplicand, the result can only be zero or a shifted version of the multiplicand: this reduces the multiplication to a series of shifts and additions, as can be seen in the example in 2.5.

$$\begin{array}{rrl}
X = & 1010 & \\
Y = & \times\quad 1100 & \\
\cline{2-2}
 & 0000 & XY_3 << 0 \\
 & 0000 & XY_2 << 1 \\
+ & 1010 & XY_1 << 2 \\
 & 1010 & XY_0 << 3 \\
\cline{1-2}
 & 1111000 & XY
\end{array}$$

Figure 2.5: Example showing the shift/add multiplication method on two four bit binary numbers.

This algorithm could be implemented simply in software, or in hardware using a k-bit adder, where k is the number of bits in the factors. However, running the algorithm would require executing between $6k + 3$ and $7k + 3$ machine instructions, depending on whether the left or right shift version of the algorithm were being used [28]. This means that even in a regular 32-bit processor, the algorithm would need to execute around 200 instructions. In a 32 bit UWRAM (with ultrawords of 1024 bits), half-ultraword multiplication would require between 3000 and 3600 instructions.

Another popular way to multiply is by using algorithms based on Booth's 1951 radix-2 algorithm for signed (twos complement) binary numbers [8], described in Algorithm 1. The recoding in this algorithm, which is achieved by looking ahead at an extra bit in each iteration, is based on Booth's observation that the additions needed to multiply a segment of consecutive ones can be replaced by one addition and one subtraction. This algorithm still requires k shifts and an average of $k/2$ additions, since it still only eliminates a single bit in each iteration, but it is the base for more efficient higher radix multiplication algorithms.

In a radix-r algorithm, $\log_2 r + 1$ bits are inspected and $\log_2 r$ bits are eliminated in each iteration. The improvement in number of iterations,however, comes at the cost of a greater number of required instructions for pre-processing and and the need for larger

Algorithm 1 Booth's radix-2 algorithm for signed binary multiplication. X is the multiplicand, Y is the multiplier,j is the number of bits in X, and k is the number of bits in Y. All new definitions of numbers have a length of $(j + k + 1)$

```
1: procedure BoothMultiply(X,Y,k)
2:     $A \leftarrow$ value of X in most significant bits
3:     $S \leftarrow$ value of -X in most significant bits
4:     $P \leftarrow Y << 1$
5:     for $c \leftarrow 1$ to $k$ do
6:         $aux \leftarrow$ two least significant bits of P
7:         if $aux = 01$ then
8:             $P \leftarrow P + A$
9:         if $aux = 10$ then
10:            $P \leftarrow P + S$
11:        $P \leftarrow P >> 1$
12:    return $P >> 1$
```

hardware components. For example, while a radix-2 algorithm requires only the calculation of $-X$ and shifted versions of X, and a two-way multiplexer for its hardware version, a radix-4 algorithm would require the calculation of $2X$, $3X$ and their complements, and a 4-way multiplexer for hardware. Despite the additional pre-processing steps, studies have shown that the high-radix approach is useful when computing the multiplication of very large numbers; for example, Javeed and Wang show that a radix-8 modular multiplication approach is faster than a radix-4 when multiplying 256 bit numbers, taking $\lceil k/3 \rceil + 4$ clock cycles to complete; this would translate to 175 cycles in a 32 bit UWRAM halfword multiplication. [23].

Other popular hardware multiplication techniques such as full-tree multipliers, which grant $O(\log k)$ multiplication for k bit integers, might prove unfeasible due to the large number of components they require. A Wallace tree multiplier, for example, requires k^2 AND gates only to generate the first level of inputs for the addition tree. In a 32 bit UWRAM, for half-word multiplication, this would be equal to 262144 AND gates.

Finally, we have the option of forgoing the idea of multiplying w^2 (or $w^2/2$) bit numbers, and instead multiplying $w/2$ fields of size w of two ultrawords to produce an ultraword with the $w/2$ results of size $2w$, ad shown in figure 2.6. This approach would clearly not require anything more than replicating the hardware necessary for w bit multiplication in a regular processor. Any of the methods described above may be used for this task. Even though microprocessor manufacturers do not reveal the algorithms or specific hardware they use to support their instruction sets, benchmarks can be run to calculate the approximate latency in cycles generated by each instruction. As an example, the latency for a 32 bit multiplication in a QuadCore Intel Core i7-7700K processor is 4.3 cycles [13].

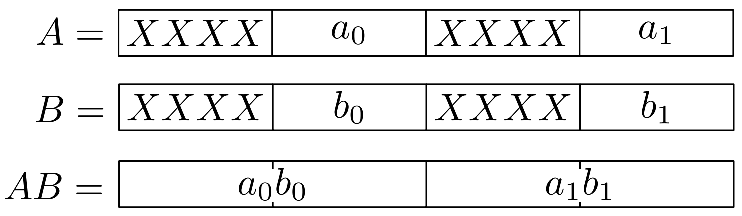

Figure 2.6: Example of multiplication of w bit numbers, with $w = 4$.

Providing this type of multiplication could be useful when calculating the multiplication of large numbers in software, using algorithms such as the Fast Fourier Transform based algorithm for numbers given in polynomial representation, which is $O(n \log n)$, where n is the degree of the polynomial.

Chapter 3

Sorting and Searching

The problems of sorting and searching are among the most studied in the field of computer science. This is because not only do they provide a very basic and fundamental functionality to computers (which is thus expected to be performed well and fast), but are also the foundation of many algorithms across different fields. In comparison based models, the complexity of sorting a list of n integers is well known to be $\Theta(n \log n)$. Fredman and Willard broke this barrier in 1990 by proposing a data structure called a fusion tree that used bit level parallelism to compare several elements at once [14]. While theoretically interesting, the constant factors associated to its operations made it unusable in practice. In this chapter we will first describe the problem of static ranking, and how it can be solved in the UWRAM, and the build on it to describe a UWRAM implementation of fusion trees.

3.1 Ranking

The definition of the rank function is as follows: given an integer q that is part of the set S, $rank(q, S) = x$ such that $S_i \leq q$ for all $i < x$. Calculating the rank of every element in the set will result in a sorted set. Hagerup proposes a word based, bit-parallel ranking algorithm in [17], which we adapt to our model and instruction set.

Elements of the set S can be represented with f bits each: f is thus the field size required to hold the representation of an element in the UWRAM. For this algorithm, we will assume that f is small enough so that all elements of S fit in a single ultraword. We will also assume that f is a power of two, such that packed elements align with the blocks of the ultraword.

Hagerup's algorithm consists of three simple steps: first, replicate the query q into every field of an ultraword and pack the elements of S into another one. Second, compare each field with the query, setting the value 1 in fields where $S_i \geq q$. Lastly, count the number of ones set to obtain the rank of q.

The first step is trivial in our model since we can simply use the replicate instruction to spread the query along an ultraword. If $f < w$, then an additional constant number of or and shift operations are needed. The elements of the set can be packed by reading from ram with a stride of w/f and shifting the fields before reading the next w elements. An example of the packing technique is shown in 3.1.

$$W_1 = \textit{spread\_sequence}\ 2 =$$
$$\boxed{00000000}\boxed{00000010}$$
$$W_2 = \textit{load\_content} \text{ from } W_1 + \textit{base} =$$
$$\boxed{00001101}\boxed{00001001}$$
$$W_2 << f =$$
$$\boxed{11010000}\boxed{10010000}$$
$$W_3 = \textit{load\_content} \text{ from } W_1 + (\textit{base} + 1) =$$
$$\boxed{00000001}\boxed{00000101}$$
$$W_2 | W_3 =$$
$$\boxed{\underbrace{1101}_{S_0}\underbrace{0001}_{S_1}}\boxed{\underbrace{1001}_{S_2}\underbrace{0101}_{S_3}}$$

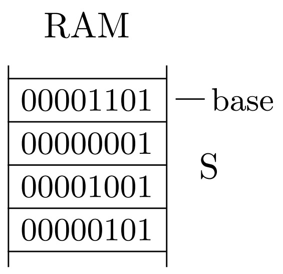

Figure 3.1: Example showing the instructions needed to pack 4 elements of $f = 4$ into an ultraword. $w = 8$ and the ultraword is shown to have only two blocks for simplicity of representation

Now that both the replication of the query and the set are contained in an ultraword each, we may compare both in a single operation, using a subtraction. The problem with this operation is that it will cause fields to exceed their boundaries when the result of the field subtraction is negative. To avoid this problem, we will extend each field with an additional bit, called a sentinel bit, which will be the most significant. The minuend, or element is extended with a value of 1 and the subtrahend (the query) with a value of 0. This trick ensures that the minuend in each field is larger than the subtrahend, and so the result will never ”leak” past field boundaries. The sentinel bit remains set to 1 if the element in that field is larger or equal to q, but is flipped to 0 if it is smaller.

Since we cannot simply expand the size of the ultraword during the execution of a program, we have two choices: either consider a larger than required field size from the beginning, such that the most significant bit can be set as required without altering the value of the element in the field, or, if this is not possible (for example when $f = w$), we can use two ultrawords instead of one for the comparisons, leaving an entire field for the sentinel bit.

Step 3 now consists simply of counting the sentinel bits that remain set to one. This can be achieved with a single *uwpopcount* instruction if $f = w$, or through the sum of the results of w/f *uwpopcount* operations when $f < w$.

Sorting S requires calculating the rank of each element in the set, and shifting it to its new position. Since calculating the rank is clearly O(1), sorting in this case is O(n).

3.2 Fusion Trees

The fusion tree data structure, proposed by Fredman and Willard in [14], adapts remarkably well to the ultra-wide model, since it can be implemented without sketching.

A fusion tree is essentially a B-tree with $B = (\log n)^{\frac{1}{5}}$ [4, 26]. The main difference between B-trees and fusion trees is the time required to search for a key within a node of the tree; while B-trees require $O(B)$ time, fusion trees can do it in $O(1)$ by comparing several elements at once. This is achieved by compressing the elements of the node into a single word so that they may be compared simultaneously. The compression is accomplished by identifying which bits are relevant to the comparison, in other words, which bits are necessary to differentiate between the elements of the node; irrelevant bits are masked out and relevant bits are packed together, such that individual bits lose their original position, but elements remain ordered. The compressed, relevant bit representation of a number is known as its *sketch*. A sketch can be computed using a prefix tree, into which each element of the node is added bit by bit, starting from the most significant. Each level of the tree l holds the values of the l^{th} bit of each element. If a level has branches, it means that it holds significant bits.

A perfect sketch cannot be computed in constant time in a restricted model, but an approximate sketch, which will include some extra zeros, can be calculated in constant time with the use of multiplication [14].

The objective of this parallelized, constant time comparison is to compute the *rank* of a query. Once the rank of a query has been calculated, either the element has been found, or the node in which to continue the search is known.

Node elements

$S_0 = 0001$

$S_1 = 0100$

$S_2 = 1000$

$S_3 = 1100$

$sketch(S_0) = 00$

$sketch(S_1) = 01$

$sketch(S_2) = 10$

$sketch(S_3) = 11$

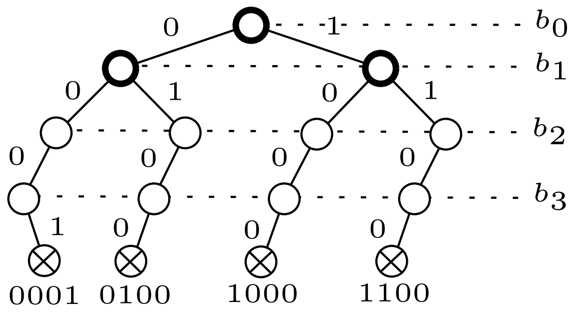

Figure 3.2: Example of the calculation of sketches for the elements of a fusion tree node.

The UWRAM has a word size of at least $w = \log n$ (or elements cannot be indexed), and an ALU that can operate on w^2 bit words at once. This means that we can use the same method for comparison that is used to calculate rank in a fusion tree node without needing any type of compression, because an ultraword can fit w elements at the time, which means that the node of a tree with a branching factor of w can be processed in constant time.

The algorithm for calculation of rank is defined in Section 3.1. For the specific case of a fusion tree node, field size $f = w$, and so the population count can be executed in a single step, since no additional alignment shifts are necessary. When the rank function is executed on an ordered set, calculating the rank also calculated the predecessor and successor of the query.

Fusion tree node =	• 0 0 0 1 • 0 1 0 0 • 1 0 0 0 • 1 1 0 0 •
	p_0 S_0 p_1 S_1 p_2 S_2 p_3 S_3 p_4
query =	0 1 0 1
Node elements extended with sentinel bits (N) =	**1** 0 0 0 1 | **1** 0 1 0 0 | **1** 1 0 0 0 | **1** 1 1 0 0
Replicated query with sentinel bits (Q) =	**0** 0 1 0 1 | **0** 0 1 0 1 | **0** 0 1 0 1 | **0** 0 1 0 1
$R = N - Q =$	**0** 1 1 0 0 | **0** 1 1 1 1 | **1** 0 0 1 1 | **1** 0 1 1 1
$uwpopcnt(\neg R) = rank(query) =$	**2**

Figure 3.3: Example of the calculation of the rank of a query in a fusion tree node.

Clearly the blocks of an ultraword cannot simply be expanded by one bit: instead, a whole block is given for each sentinel bit, and the calculation is split into two parts.

To sort a sequence of n elements with a fusion tree, all elements must be inserted into the tree. The time to insert an element depends directly on the height of the tree and on the time needed to find an element within a node. Since a fusion tree is in essence a B-tree, its height is $O(\log_B n)$ with $B = \log n$, so the height of a fusion tree is proportional to $O(\frac{\log n}{\log \log n})$. Since the time to search within a node is $O(1)$, the complexity of searching or inserting an element is $O(\frac{\log n}{\log \log n})$, and the complexity of sorting is $O(n\frac{\log n}{\log \log n})$.

Even though the use of ultrawords does not improve the performance of the fusion tree asymptotically, it does improve the constant factors drastically. It is well known that a data structure such as this one has only theoretical value in traditional architectures due to the large coefficients associated to its operations [37], but an architecture such as this one could make it usable.

Another less practical way of adapting the fusion tree to the UWRAM model is by modifying the branching factor to $B = (\log^2 n)^{\frac{1}{5}}$, preserving the need for *sketch* and *desketch* functions, but achieving a structure with operations that can be preformed in $O(\frac{\log n}{\log \log^2 n})$ time, with large constant factors, in a UWRAM model that supports ultraword sized multiplication.

3.3 Predecessor Search

The predecessor of a query q in a set S can be defined as follows:

$$predecessor(q, S) = max\{s \in S | s \leq q\}$$

This problem is important because it is an integral part of sorting and searching in general. If the query q exists in S, the predecessor search is simply the search for q; if q is not an element of S, then a predecessor search will tell us where to insert it.

In their 2006 paper [30], Patrascu and Thorup explore the time and space trade-offs for predecessor search in different models of computation. They describe a RAM based model with word size b, where elements of the set S can be represented with l bits and $b = Bl$ (a standard word RAM has $b = l$). This model describes the UWRAM perfectly: b is the size of an ultraword (which can be considered word size, since it can be read from memory in unit time),l is the regular word size w and B is also w, giving us the definition of an ultraword $u = w^2$.

They show that the optimal predecessor search time in this model is given by $\log_b n = \frac{lg(n)}{lg(B)+lg(n)} = \Theta(min\{\log_l n, log_B n\})$ which are the running times of predecessor search in fusion trees and B-trees respectively. Since in our model $l = B = w = \log n$, the optimal search time is given by $O(\frac{\log n}{\log \log n})$, unless there is a better RAM algorithm that takes no benefit from the larger word size.

Chapter 4

Simulations

In order to provide a more empirical perspective of the UWRAM model, we decided to test some algorithms in the UWRAM and PRAM models to see how they compared. Since neither machine exists in practice, we will test them through simulations. We chose the number of instructions executed to be our measure of performance; since we assume all instructions take the same time to execute, this measure can give us a good idea of how long it would take to execute a program in either model.

For clarity, we will refer to the programs which simulate the machines, RAM and assembler as the simulators, and to the programs we run in the simulators as simulations. There are thus two simulators: one for the PRAM, and one for the UWRAM model. Each receive an assembly style program as input, which is then run on a simulated RAM and registers, while counting the number of instructions needed to end the execution flow of the program. The instruction sets used for the simulators are based on the MIPS RISC instruction set, since the reduced instruction set is similar to the one we wished to implement, and the register to register syntax is simple to read and write [34]. The complete instruction sets used can be found in Appendix A. All simulations are written in the language described by this instruction set. Execution stops when the program has finished running in case of the UWRAM, and when all processors have finished their execution in the PRAM. Instructions are counted in parallel in the PRAM, which means that if each processor executes an instruction simultaneously, it is counted as one instruction for the final instruction count.

Specifically, we will simulate 32 bit UWRAM, with words of 32 bits and two types of registers: regular registers (32 registers of 32 bits), and ultra wide registers (32 registers of 1024 bits). The PRAM will have a word size of 32 bits, and 32 processors with 32 registers of 32 bits each.

We decided to test two different types of problems; the first one, known as the subset sum problem, is generally solved using dynamic programming techniques, and is well known to be difficult to parallelize. The second is the string matching problem, which is very simple to parallelize in multi-core models.

4.1 The Subset Sum Problem

The subset sum problem can be stated as follows: given a set $S = \{s_1, s_2, ..., s_n\}$ of positive integers and a positive integer t, find a subset S' of S such that $\sum_{S'} s_i \leq t$ and is maximized. The decision version of the problem, which is the one we will work with in this section, asks whether there is a subset S' such that $\sum_{S'} s_i = t$. The algorithms we will use for these simulations are based on the dynamic programming (DP) solution proposed by Bellman in [5] , which means that in order to calculate the answer for the decision problem for a target sum of t, the algorithm must also calculate the answer for every $t' < t$; this means that once the decision is given, either the answer is $true$ or a simple iteration through the last row of the DP table is the only step required to obtain the largest t' attainable with the given set. For simplicity we will use the basic decision version of the problem, which returns a $true$ or $false$ answer.

These algorithms are a great example of techniques that can benefit greatly from bit parallelism and not as much from multiprocessor parallel design. Filling each cell of the DP table requires access to elements in the previous row (which must have been filled already) and elements in the same row where the cell is located, which means that DP algorithms are essentially sequential and the table cannot be split in any trivial way to be filled in parallel. In the following sections we will review two algorithms which are based on Bellman's original DP algorithm: one is adapted to use the extensive bit parallelism of the UWRAM, and the other to fit a multicore architecture such as that of the PRAM.

4.1.1 The UWRAM Algorithm

For the UWRAM solution we will use the algorithm proposed in [11] which is a direct implementation of Pisinger's Word RAM algorithm [31] in the UWRAM. The algorithm uses the following recursion:

$$C_i = (C_{i-1} \mid C_{i-1} >> s_i)$$

where C_i is the i^{th} row of the DP table. Each bit position in the row represents a different value of t: the bit will be 1 if that sum can be achieved with the elements of S inspected so far, and 0 otherwise. This representation allows w^2 elements of the table to be updated simultaneously and in constant time in the UWRAM. Shifting a row by s_i is the same as adding s_i to each element already in the row. This algorithm runs in $O(nt/\log^2 t)$ time, which is proportional to the number of ultrawords in the table.

Algorithm 2 UWRAM algorithm for the subset sum problem

```
procedure isSubsetSum(S,T)
    $w \leftarrow$ ultraword size in bits
    $words\text{-}per\text{-}row \leftarrow \lceil (t+1)/w \rceil$
    $rows \leftarrow size(S) + 1$
    $dptable[rows][words\text{-}per\text{-}row]$
    $dptable[0][0] \leftarrow 1 << (w-1)$        ▷ Place 1 in the first position of the first row
    for $c \leftarrow 1$ to $rows$ do
        $aux \leftarrow dptable[c-1] >> S[c-1]$
        for $i \leftarrow 0$ to $words\text{-}per\text{-}row$ do
            $dptable[c][i] \leftarrow dptable[c-1][i] | aux[i]$
    $mask \leftarrow 1$        ▷ Ultraword with 1 in the last block
    $result \leftarrow dptable[rows-1][words\text{-}per\text{-}row - 1] >> (w * words\text{-}per\text{-}row - t - 1)$
    return $result$ & $mask$
```

4.1.2 The PRAM Algorithm

For the PRAM solution we will use the first algorithm proposed by Sanches, Soma and Yanasse in [32]. Their algorithm runs in $O(\frac{n}{p}(t - s_{min}))$ on a CREW PRAM, with the restriction that the number of processors $p \leq s_{min}$. They present two other algorithms with the same asymptotic run time, but one requires that $p \leq n$ and requires processors to make queries to RAM cells to check if they are available for writing, and the other requires that $\log(n - 2\log t) \leq p \leq n - \log t$. For simplicity, we will describe and test only the first algorithm. Other examples of parallel algorithms for the subset sum can be found in [25, 7, 10] but none of them improve on the $O(\frac{n}{p}(t - s_{min}))$ time, even when using a CRCW PRAM.

Sanches' algorithm fills a vector g of size $t - s_{min}$ instead of the entire DP table, and the vector is divided into groups of size p, that are processed in parallel. An extra $d = s_{max} - s_{min} - 1$ positions are needed in the vector to allow required accesses when processing the first block. At the end of the execution, target values in the vector that cannot be achieved with the given set hold a value of n, otherwise, they hold the index of the largest element in the set needed for the solution.

The algorithm is divided into three stages or phases. In the first, the vector is initialized with the value n. In the second, solutions with a single element are allocated, and in the third, the rest of the vector is filled. This is done by checking each element of the set against each position of the vector, in increasing order. The detailed algorithm is described below.

Algorithm 3 PRAM algorithm for the subset sum problem

```
1: procedure isSubsetSum(S,t)
2:     $d \leftarrow s_{max} - s_{min} - 1$
3:     $g[t + d + 1]$
4:     in parallel:
5:     for $k \leftarrow 0$ to $\lceil (t + d)/p \rceil - 1$ do                      ▷ Phase I
6:         if $kp + pid - d \leq t$ then
7:             $g[kp + pid - d] \leftarrow n$
8:     for $k \leftarrow 0$ to $\lceil (t - s_{min})/p \rceil - 1$ do                 ▷ Phase II
9:         $t_{pid} \leftarrow kp + pid + s_{min}$
10:        if $t_{pid} \leq t$ then
11:            for $j \leftarrow n - 1$ to $0$ do
12:                if $s_j = t_{pid}$ then
13:                    $g[t_{pid}] \leftarrow j$
14:    for $k \leftarrow 0$ to $\lceil (t - s_{min})/p \rceil - 1$ do                 ▷ Phase III
15:        $t_{pid} \leftarrow kp + pid + s_{min} + 1$
16:        if $t_{pid} \leq t$ then
17:            for $j \leftarrow n - 1$ to $0$ do
18:                if $(g[t_{pid} > j)]$and$(g[t_{pid} < j])$ then
19:                    $g[t_{pid}] \leftarrow j$
20:    return $g[t - 1]$
```

4.1.3 Experiments and Results

Since the runtimes of both algorithms depend on the values of both t and n, we will run two separate tests on each of these values. In the first test, $n = 20$ is chosen arbitrarily and remains constant throughout the experiment, while the values of t range from 1000

t	**UWRAM instructions**	**PRAM instructions**
1000	1170	14012
5000	2565	72047
10000	4395	146328
50000	17665	731135
100000	34320	1467553
200000	68105	2891266
300000	103080	4163228
400000	134525	5719068
500000	176005	7245660
600000	205820	8672044
700000	241690	9873753
800000	275365	11755944
900000	309685	13104753
1000000	360840	14529597

Table 4.1: Test values and results for subset sum tests on the value of t.

to $1,000,000$. In the second test, $t = 100,000$ is chosen arbitrarily and the values of n range from 10 to 1000. For both tests, the values of the elements of S are random numbers between 32 and t. Even though both algorithms make use of multiplication, which the models do not allow, it is always by the same value (w in the UWRAM algorithm and p in the PRAM), and so it can be replaced by a left shift, since both of these values are powers of two, which means that if $x = 2^k$, then $xy = y << k$. Integer division can be implemented the same way with left shifts. In our programs, k is given as a parameter; since most parallel programs need to be aware of the number of processors available or the word size in case of bit parallelism, it would also be reasonable to add an instruction to copy this number to a register, so that it would not be necessary to hard code it into programs.

The values used and results obtained for the first experiment can be found in Table 4.1.

The values used and results obtained for the second experiment can be found in Table 4.2. As we can see from the results of both experiments, the UWRAM algorithm runs approximately 38 times faster than its PRAM counterpart.

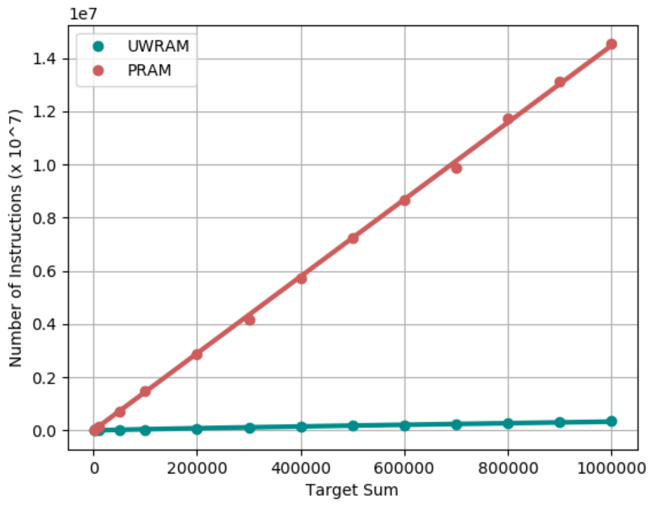

Figure 4.1: Number of instructions needed for the execution of the subset sum algorithms with increasing values of the traget sum.

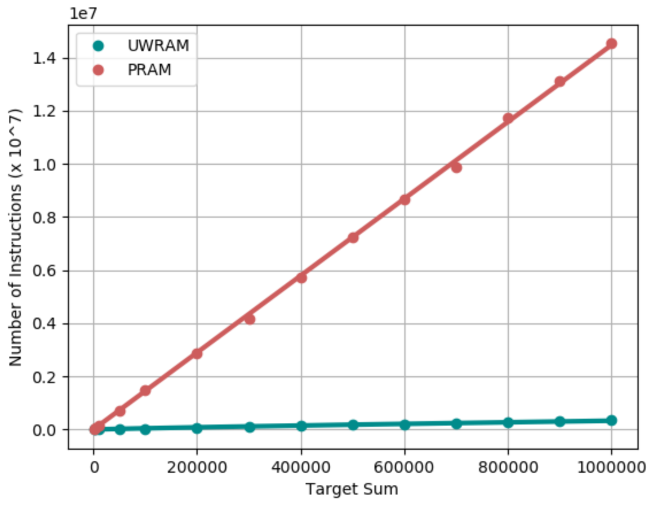

Figure 4.2: Number of instructions needed for the execution of the subset sum algorithms with increasing values of n.

n	**UWRAM instructions**	**PRAM instructions**
10	17025	672972
50	89135	3439830
100	173300	6705313
200	353305	13326184
300	527645	19911756
400	702790	26764199
500	875440	33366198
600	1054430	40225810
700	1229810	46889054
800	1405945	53833329
900	1577860	60271379
1000	1744990	66898268

Table 4.2: Test values and results for subset sum tests on the value of n.

4.2 String Matching

Given a text T of size n and a pattern P of size m, both taken from an alphabet Σ of size σ, the string matching problem asks for the indices of all the occurrences of P in T. Generally, n is assumed to be much larger than m: for this reason we will look at algorithms that allow pre-processing of the pattern, but not of the text. Unlike the subset sum example, this problem is very easy to parallelize; to parallelize across p processing units, all that is required is to split the text into p parts, so that each processor may search its assigned segment independently. The search can be parallelized the same way across the blocks of the ultraword in the UWRAM. In this case, we will use the same algorithm for both models.

4.2.1 The Boyer-Moore-Horspool Algorithm

The algorithm we will test on both models is a simplification of Boyer and Moore's 1977 string searching algorithm [9] proposed by Horspool in 1980 [19]. Boyer and Moore's algorithm is based on the observation that more information can be obtained from comparing the last character of the pattern to the text than when comparing the first. The pattern is aligned with the text and the last position of the window is checked; if the last character of the text window is not present in the pattern, clearly there is no possible alignment of the pattern and the text that will produce a match and include that character. This means that the algorithm can safely skip m characters in the text without having ever inspected them, which results in a very fast algorithm in the average case. The algorithm performs a preprocessing step on the pattern to generate two tables: *delta1*, which holds an entry for each character in the alphabet indicating how many characters the algorithm may skip if said character is found at the end of the window being inspected, and *delta2*, which holds an entry for each position of the pattern at which a mismatch may occur (for cases where the pattern is partially matched), and is built by identifying possible reocurrences of suffixes of the pattern within the pattern itself. Horspool's main observation and change to the algorithm is that in most cases, the second table results in very small improvements when compared to the first, so it may be omitted by making a small change to *delta1*, which consists of saving index of the second to last occurrence of the last character in the pattern, instead of the last. Horspool names this modified table *delta12*. A description of the Boyer-Moore-Horspool (BMH) algorithm is given in Algorithm 4.

Algorithm 4 Boyer-More-Horspool algorithm for string matching.

```
procedure BMH(TEXT,N,PATTERN,M)
    $delta12[*] \leftarrow m$                          ▷ Build delta12 table
    for $j \leftarrow 0$ to $m-2$ do
        $delta12[*] \leftarrow m-j$
    $lastch \leftarrow Pattern[m-1]$                   ▷ Last character of Pattern
    $i \leftarrow m-1$
    while $i < n$ do
        $ch \leftarrow Text[i]$
        if $ch = lastch$ then
            if $Text[i-m...j] = Pattern$ then
                save $i-m$
        $i \leftarrow i + delta12[ch]$
```

The parallelization of the algorithm in the PRAM model is straight-forward: the calculation of the *delta12* table is performed sequentially, and then the text is split into p segments of equal size, so that each processor may execute the BMH algorithm on a single segment independently.

We adapt the algorithm to the UWRAM in the same way as in the PRAM; each block of the ultraword will simulate a PRAM processor, and so each block will process a segment of text of size $(n/w)+(m-1)$, each segment will have an overlap of $m-1$ characters with the following segment, to ensure that no window of text is left unchecked. The *delta12* table is calculated sequentially in the same way as in the PRAM algorithm.

Several considerations need to be made in order to simulate the independence between blocks of the ultraword. First, it is necessary to calculate how many blocks we need to fit the pattern by calculating $b = (\log \sigma)m/w$. In the examples used in our simulations, $b = 1$. This case is the simplest in the UWRAM, because it allows for natural memory accessing, where each block keeps track of the address of the text it is searching, and updating the addresses is simple, as we will describe ahead.

In order to compare windows of text with the pattern simultaneously, we must pack the pattern and then replicate it across an ultraword. Since we assume the entire pattern fits in a block, this can be done by packing the pattern into a simple register and then spreading it using the spread instruction (UWSPR). The first set of text addresses (The addresses of the beginning of each text segment) can be calculated by spreading a sequence of the size of segments (UWSSQ). Now the separate windows of the pattern may be packed into a single ultraword and compared with the sentinel bit technique described in chapter 3. A sentinel bit need only be set at the beginning of each block, and not at the beginning of every character, since we are only searching for exact matches, and do not care where the first mismatch occurs. If the packing is tight and does not allow for a sentinel to be added, the comparison can be done in two steps instead, by giving an entire block to the sentinel.

In order to identify the windows where there is an exact match of the pattern, it is necessary to perform two subtractions, $P - T$ and $T - P$, where P is the ultraword holding the replicated pattern and T is the ultraword holding the windows of text. Performing a logical AND on the result of of both operations will result in a one in the most significant bit of each block where the pattern has been found. A compress operation can be used to check if any of the sentinel values are set to one. If the compression equals zero, then the pattern was not found in any block, and the addresses may be updated. If the compression is different to zero, then the pattern has been found in at least one block and a mask to select the relevant values from the address word can be calculated by computing $M = H - (H >> (f - 1))$, where H is the result of ANDing the result of both subtractions, with all but the sentinel bits masked out, and f is the field size [11].

The addresses can be updated the following way: first, all but the last character of each block are masked out, so that an access by content, with the base address of the *delta12* table as the base will result in $delta12[last_i]$ in each block, where $last_i$ is the last character of the text window in block i. The values obtained can now be added to the original set of addresses, and the result will be the updated set of addresses, ready for the next iteration of the search loop.

The algorithm has an average runtime of $O(n)$ in both models ($O(n/p)$ for the PRAM and $O(n/w)$ in the UWRAM). In the worst case, which is when the pattern can be found at every index of the text, the PRAM version has a running time of $O(nm)$. In the worst case, provided that the pattern can be fitted into a single block, the run time for the UWRAM remains $O(n)$, since comparisons are made on the whole pattern at once. In fact, as long as the pattern can fit into an ultraword, the runtime remains $O(n)$, although with no parallelization on the text, the actual runtime would be much slower than that of the PRAM. If the pattern exceeds the size of an ultraword, then the worst case runtime is $O(mn \log \sigma / w^2)$.

4.2.2 Experiments and Results

We will run two separate tests on varying values of n. In both tests, the value of m will remain constant, and will be such that the entire pattern can be packed into a single block of the ultraword. Since we will be using an 8 bit ascii alphabet, and a 32 bit UWRAM model, m will be equal to four. We will test two types of text: the first will be a "lorem ipsum" Latin placeholder text generated with an online tool [1], and for which the pattern will be an arbitrary four character pattern taken from the text. The second text will be an example of a worst case text, and will be a long string of a single ascii character, with a pattern of four instances of that same character. Both texts will be tested in a range of $1,000$ to $100,000$ characters. As in the previous experiment, we will use number of instructions executed as a measure of performance.

The values used and results obtained from the first experiment can be found in Table 4.3 and Figure 4.3. The values used and results obtained from the second experiment can be found in Table 4.4 and Figure 4.4.

Fitting the first experiment's results to a straight line gives a linear equation of $0.27n + 840.24$ for the UWRAM and $0.099n + 876.44$ for the PRAM. For the second experiment, the equations are $1.09n + 807.66$ and $1.50n + 875.56$ respectively. Even though the UWRAM is approximately 2.6 times slower than the PRAM in the average case, it is not a bad result, considering we are comparing the performance of a single processor against 32. To truly be able to choose the superior model (regarding this problem), it would be necessary to analyze the cost of each one.

n	**UWRAM instructions**	**PRAM instructions**
1000	1083	984
5000	2148	1404
10000	3513	1876
20000	6243	2896
30000	8863	3833
40000	11598	4870
50000	14278	5802
60000	16893	6809
70000	19473	7704
80000	22268	8844
90000	24923	9903
100000	27633	10850

Table 4.3: Test values and results for string matching tests using a regular text on the value of n.

n	**UWRAM instructions**	**PRAM instructions**
1000	1903	2386
5000	6278	8386
10000	11738	15874
20000	22693	30849
30000	33613	45874
40000	44568	60849
50000	55488	75874
60000	66443	90849
70000	77363	105874
80000	88318	120849
90000	99238	135874
100000	110193	150849

Table 4.4: Test values and results for string matching tests using a regular text on the value of n.

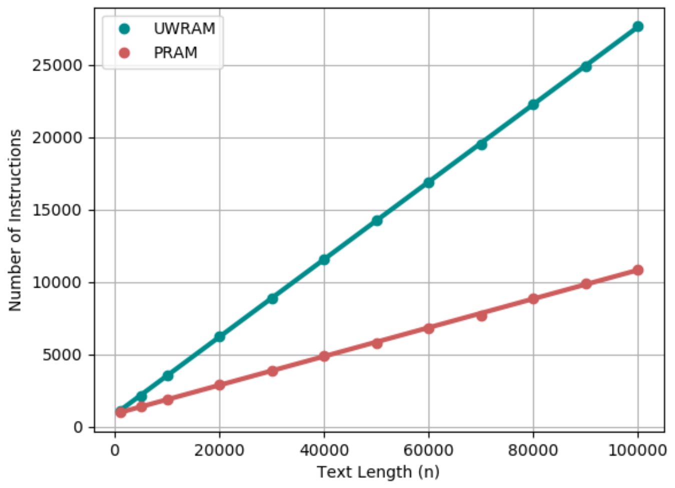

Figure 4.3: Number of instructions needed for the execution of the BMH algorithm with increasing values of n, in an average case.

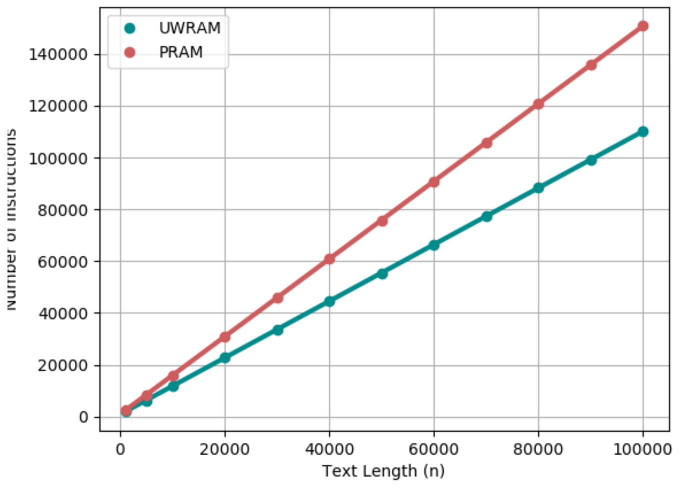

Figure 4.4: Number of instructions needed for the execution of the BMH algorithm with increasing values of n, in the worst case.

Chapter 5

Conclusions

In this work, we have explored the Ultra-Wide Word model proposed by Farzan, Lopez-Ortiz, Nicholson and Salinger in [11] in a more empirical fashion. In Chapter 2 we describe the model and provide additional instructions to ease the parallelization of different procedures, such as packing of small fields in an ultraword and sentinel bit arithmetic, both which are key to the efficient parallelization of different algorithms. We also discuss some of the weaknesses of the model, in particular its lack of multiplication support, and several ways in which it could be implemented (out of the unit cost assumption).

In Chapter 3, we discuss the problem of sorting and searching in the model, and show that searching can be achieved in $O(\frac{\log n}{\log\log n})$ time , and sorting in $O(n\frac{\log n}{\log\log n})$ time with the use of fusion trees, which can be implemented in the model in an efficient way with low constant factors.

In Chapter 4, we present the results of simulating the algorithms to solve two different problems in both the UWRAM and the PRAM. The subset sum problem is solved in both cases with a modified dynamic programming algorithm, adapted to the different types of parallelism of the models. We showed that for this problem, which is traditionally hard to parallelize in multi-core models, the UWRAM significantly outperformed the PRAM in number of instructions needed to complete the task. The second simulated problem was string matching; here we show that even when dealing with problems that are easy to parallelize in a multi-core model, the UWRAM can perform reasonably well. In an average case, the PRAM is around 2.6 times faster (requires 2.6 times less instructions) than the UWRAM, and in the worst case, the UWRAM runs approximately 1.4 times faster.

We have shown a wide range of techniques to aid in the bit-parallelization of algorithms in the model, and how many algorithms and data structures may be adapted for UWRAM use. We also show through simulation that the UWRAM can perform well practically even when compared with a PRAM, which is a stronger parallel model. The logical next step for this research would be to create a hardware model for it, in order to determine aspects like pricing more precisely, so that its viability, and preferability against other parallel models may be determined objectively.

References

[1] Lorem ipsum. http://www.lipsum.com/, 2017. Accessed: 2017-07-17.

[2] Blaise Barney. Introduction to parallel computing. https://computing.llnl.gov/tutorials/parallel-comp, 2017.

[3] Youssef Bassil and Aziz Barbar. Sequential and parallel algorithms for the addition of big-integer numbers. *CoRR*, abs/1204.0232, 2012.

[4] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

[5] Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[6] Philip Bille. Fast searching in packed strings. *Journal of Discrete Algorithms*, 9(1):49 – 56, 2011. 20th Anniversary Edition of the Annual Symposium on Combinatorial Pattern Matching (CPM 2009).

[7] Saniyah S. Bokhari. Parallel solution of the subset-sum problem: an empirical study. *Concurrency and Computation: Practice and Experience*, 24(18):2241–2254, 2012.

[8] Andrew D. Booth. A signed binary multiplication technique. *The Quarterly Journal of Mechanics and Applied Mathematics*, 4(2):236–240, 1951.

[9] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, October 1977.

[10] V. V. Curtis and C. A. A. Sanches. An efficient solution to the subset-sum problem on gpu. *Concurrency and Computation: Practice and Experience*, 28(1):95–113, 2016. cpe.3636.

[11] Arash Farzan, Alejandro López-Ortiz, Patrick K. Nicholson, and Alejandro Salinger. Algorithms in the ultra-wide word model. *CoRR*, abs/1411.7359, 2014.

[12] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21-9, 1972.

[13] Agner Fog. x86,x64 instruction latency, memory latency and cpuid dumps. http://users.atw.hu/instlatx64/, 2017. Accessed: 2017-07-17.

[14] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424 – 436, 1993.

[15] Kimmo Fredriksson and Szymon Grabowski. Exploiting word-level parallelism for fast convolutions and their applications in approximate string matching. *European Journal of Combinatorics*, 34(1):38 – 51, 2013. Combinatorics and Stringology.

[16] Julio Gea-Banaloche and Laszlo B. Kish. Future directions in electronic computing and information processing. *Proceedings of the IEEE*, 93-10, 2005.

[17] Torben Hagerup. *Sorting and searching on the word RAM*, pages 366–398. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.

[18] Pasi Hämäläinen. A pram emulator, 1992.

[19] R. Nigel Horspool. Practical fast searching in strings. *Software: Practice and Experience*, 10(6):501–506, 1980.

[20] Intel. *Intel SSE4 Programming Reference*, 7 2007.

[21] Intel. *Intel Advanced Vector Extensions Programming Reference*, 6 2011.

[22] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, 7 2017.

[23] Khalid Javeed and Xiaojun Wang. Speed and area optimized parallel higher-radix modular multipliers. *IACR Cryptology ePrint Archive*, 2016:53, 2016.

[24] Eric E. Johnson. Completing an mimd multiprocessor taxonomy. *SIGARCH Comput. Archit. News*, 16(3):44–47, June 1988.

[25] Vipin Kumar. *Introduction to Parallel Computing.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[26] Luis A. A. Meira and Rogério H. B. de Lima. Fusion tree sorting. *CoRR*, abs/1411.0048, 2014.

[27] Nvidia. Quadro dor desktop workstations: products, features and specifications. `http://www.nvidia.ca/object/compare-quadro-gpus.html`. Accessed: 2017-07-17.

[28] Behrooz Parhami. *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, Oxford, UK, 2000.

[29] Behrooz Parhami. *Introduction to parallel processing: algorithms and architectures*. Springer Science and Business Media, 2006.

[30] Mihai Patrascu and Mikkel Thorup. Time-space trade-offs for predecessor search. *CoRR*, abs/cs/0603043, 2006.

[31] Pisinger. Dynamic programming on the word ram. *Algorithmica*, 35(2):128–145, Feb 2003.

[32] C.A.A. Sanches, N.Y. Soma, and H.H. Yanasse. Parallel time and space upper-bounds for the subset-sum problem. *Theoretical Computer Science*, 407(1):342 – 348, 2008.

[33] Dezs Sima. Decisive aspects in the evolution of microprocessors. *Proceedings of the IEEE*, 92-12, 2004.

[34] Dominic Sweetman. *See MIPS Run*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[35] Henry S. Warren. *Hacker's Delight*. Addison-Wesley Professional, 2nd edition, 2012.

[36] Wikipedia. Comparison of intel processors. `https://en.wikipedia.org/wiki/Comparison_of_Intel_processors`. Accessed: 2017-07-20.

[37] Dan E. Willard. Examining computational geometry, van emde boas trees, and hashing from the perspective of the fusion tree. *SIAM Journal on Computing*, 29(3):1030–20, 2000. Copyright - Copyright] 2000 Society for Industrial and Applied Mathematics; Last updated - 2012-06-29.

APPENDICES

Appendix A

Instruction Set for Simulators

The instruction set for the PRAM simulator can be found in Table A.1, and the instruction set for the UWRAM con be found in Tables A.1 and A.2. Regular registers begin with \$r and ultrawide registers begin with \$u.

Table A.1: Basic Register Instruction Set

Instruction	**Syntax**	**Meaning**
ADD	ADD \$rd, \$r1, \$r2	\$rd = \$r1 + \$r2
ADDI	ADDI \$rd, \$r1, imm	\$rd = \$r1 + imm
AND	AND \$rd, \$r1, \$r2	\$rd = \$r1 ∧ \$r2
ANDI	\$rd, \$r1, imm	\$rd = \$r1 ∧ imm
BEQ	\$r1, \$r2, off	if(\$r1 == \$r2) then branch to off
BNE	BNE \$r1, \$r2, off	if(\$r1 != \$r2) then branch to off
J	J add	PC = add
JAL	JAL add	\$r29 = PC, PC = add
JR	JR \$r1	PC = \$r1
LI	LI \$rd, imm	\$rd = imm
LUW	LUW \$rd, \$u1	\$rd = \$u1[0]
LW	LW \$rd, \$r1, off	\$rd = MEM[\$r1 + off]
NOT	NOT \$rd,\$r1	\$rd = ¬\$r1
OR	OR \$rd, \$r1, \$r2	\$rd = \$r1 ∨ \$r2
ORI	ORI \$rd, \$r1, imm	\$rd = \$r1 ∨ imm
SL	SL \$rd, \$r1, \$r2	\$rd = \$r1 << \$r2
SLI	SLI \$rd, \$r1, imm	\$rd = \$r1 << imm
SLT	SLT \$rd, \$r1, \$r2	\$rd = (\$r1 < \$r2)

Instruction	Syntax	Meaning
SLTI	SLTI \$rd, \$r1, imm	\$rd = (\$r1 < imm)
SR	SR \$rd, \$r1, \$r2	\$rd = \$r1 >> \$r2
SRI	SRI \$rd, \$r1, imm	\$rd = \$r1 >> imm
SUB	SUB \$rd, \$r1, \$r2	\$rd = \$r1 - \$r2
SW	SW \$r1, \$r2, off	MEM[\$r2 + off] = \$r1
XOR	XOR \$rd, \$r1, \$r2	\$rd = \$r1 ⊕ \$r2
XORI	XORI \$rd, \$r1, imm	\$rd = \$r1 ⊕ imm

Table A.2: Ultra-Wide Register Instruction Set

Instruction	Syntax	Meaning
UWADD	UWADD \$ud, \$u1, \$u2	\$ud = \$u1 + \$u2
UWAND	UWAND \$ud, \$u1, \$u2	\$ud = \$u1 ∧ \$u2
UWBEQ	UWBEQ \$u1, \$u2, off	if(\$u1 == \$u2) then branch to off
UWBNE	UWBNE \$u1, \$u2, off	if(\$u1 != \$u2) then branch to off
UWCOM	UWCOM \$ud, \$u1	\$ud[0][j] = \$u1[j][0]
UWEXP	UWEXP \$ud, \$u1	\$ud[j][0] = \$u1[0][j]
UWLB	UWLB \$ud,j,base	\$ud[j] = MEM[base+j]
UWLC	UWLC \$ud,\$uc,base	\$uc[j] = MEM[base+\$ud[j]]
UWLRR	UWLRR \$ud, \$r1	\$ud[0] = \$r1
UWLUW	UWLUW \$ud,base	\$ud[j] = MEM[base+j]
UWNOT	UWNOT \$ud,\$u1	\$ud = ¬\$u1
UWOR	UWOR \$ud, \$u1, \$rs	\$ud = \$u1 ∨ \$u2
UWSB	UWSB \$ud,j,base	MEM[base+j] = \$ud[j]
UWSC	UWSC \$ud,\$u1,base	MEM[base+\$u2[j]] = \$u1[j]
UWSL	UWSL \$ud, \$u1, \$r1	\$ud = \$u1 << \$r1
UWSLI	UWSLI \$ud, \$u1, imm	\$ud = \$u1 << imm
UWSPR	UWSPR \$ud, \$u1	\$ud[j] = \$r1 for all j
UWSR	UWSR \$ud, \$u1, \$r1	\$ud = \$u1 >> \$r1
UWSRI	UWSRI \$ud, \$u1, imm	\$ud = \$u1 >> imm
UWSRR	UWSRR \$rd, \$u1	\$rd = \$u1[0]
UWSSQ	UWSSQ \$ud,imm	\$ud[j] = j*imm
UWSSQR	UWSSQR \$ud, \$r1	\$ud[j] = j*\$r1
UWSUB	UWSUB \$ud, \$u1, \$rs	\$rd = \$r1 - \$rs
UWSUW	UWSUW \$ud,base	MEM[base+j] = \$u1[j]
UWXOR	UWXOR \$ud, \$u1, \$u2	\$ud = \$u1 ⊕ \$u2