WM FIFOS: Size Analysis

Wm. A. Wulf
Rohit V. Wad

Computer Science Report No. TR-91-16
July 5, 1991

Contents

Abstract

The WM computer architecture interposes data FIFOs between the execution units and memory. Data FIFOs improve performance by permitting memory delays to be overlapped with instruction execution. The depth of a FIFO depends on the average rate of memory accesses and the proximity of references to the FIFO. This project aims at exploring the effect of FIFO depth on performance, and suggesting a size that would be suitable for most applications.

Acknowledgments

I would like to thank my advisor, Prof. Wm. A. Wulf, and Prof. Anita Jones for helping me explore the depths of .... the WM FIFOs. Thanks also to Ricky Benitez for answering my questions on the WM compiler and to Ramesh Peri for the "designer" 4!

1 Introduction

The WM computer architecture interposes data FIFOs between the execution units and memory. These FIFOs serve to hide memory latency from the execution units by allowing memory delays to be overlapped with instruction execution. It is the aim of this report to explore the effect of the data FIFO depths on performance.

Background Information

The WM computer architecture is a high performance scalar architecture with roughly the hardware complexity associated with RISC machines, and roughly the code densities associated with CISC machines [8]. WM is designed to exploit concurrency at several levels. The architecture achieves high performance by using multiple asynchronous functional units, as in [7]. The WM architecture decouples data access from execution, i.e., access to memory to fetch and store results proceeds in parallel with instruction execution. This is quite similar to the decoupled access/execute computer architectures (DEA) described in [6].

The basic structure of the WM architecture is shown in Figure 1. It consists of a memory system, an instruction fetch unit (IFU), and integer execution unit (IEU), and a floating point execution unit (FEU). The execution units can operate in parallel in the absence of data dependencies.

1.1. The Execution Units

The execution units run under the control of the instruction fetch unit, IFU. The IFU enqueues instructions for execution by each of the execution units in a set of FIFOs, one

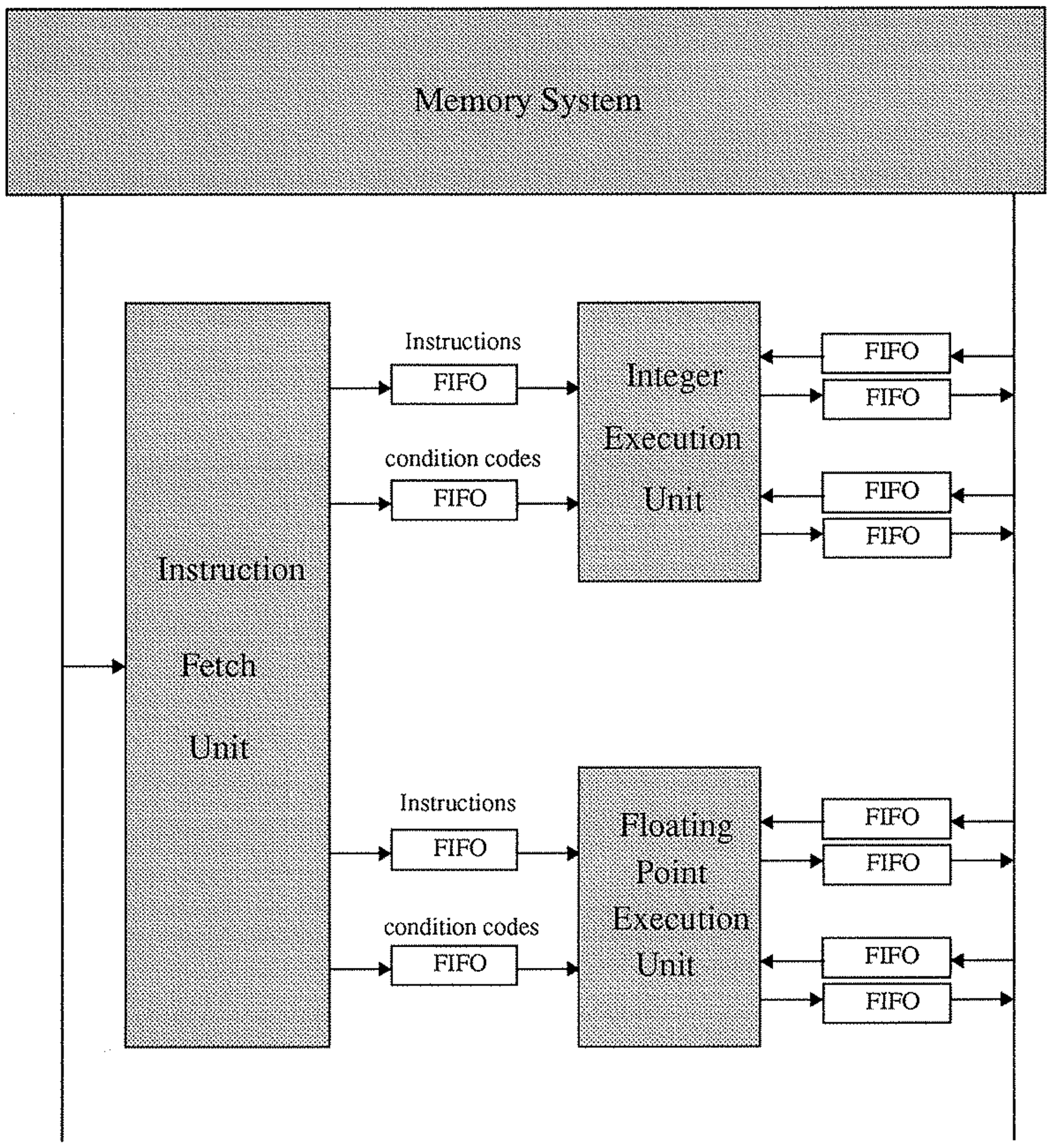


Figure 1: WM — Basic Machine Structure

for each unit. This allows concurrent execution of instructions by the execution units. The execution units can "slip" away from each other, as in [4], and can be executing instructions from different areas of the code, while still maintaining the semantics of sequential execution.

Each execution unit executes instructions of the form:

dst:= (src1 **op1** src2) **op2** src3

That is, an instruction specifies a destination register, three source operands, and two operators. A source operand may be the contents of a register, an unsigned literal, or the contents of an input FIFO. The destination may be a register, or an output FIFO.

The IEU and the FEU are implemented as a pair of pipelined ALUs, as shown in Figure 1.1. While ALU2 is executing the second operation (op2) of an instruction, ALU1 might be executing the first operation (op1) of the succeeding instruction. Thus two operations can be executed in each cycle.

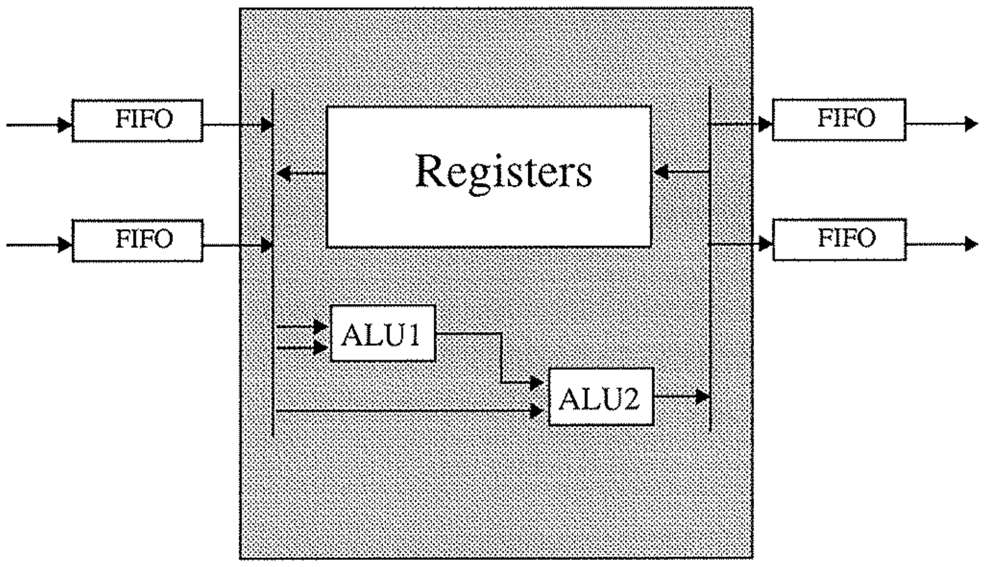


Figure 1.1: Execution Unit Configuration

Each of the execution units has its own register set. Thus the floating point register set is distinct from the integer register set. The floating point instructions refer to registers in the FEU, and the integer instructions refer to registers in the IEU. It is important to note that integer instructions are used for address computations and to specify data transfers between memory and registers of both types.

As discussed in this section, the IEU and the FEU can refer to registers, FIFOs and unsigned literals as source operands. In order to read from or write to the memory, the execution units have to go through FIFOs. The memory operations are discussed in the next section.

1.2. The Memory System

WM uses an unconventional method to perform memory reads and writes. It uses FIFOs between the execution units and the memory. The execution units refer to "register 0" to name the input and the output FIFOs, depending on the context.

An instruction can refer to register 0 as a source operand to dequeue data from the input FIFO, and refer to it as a destination operand to enqueue data into the output FIFO. Loads and Stores are always executed by the IEU, even thought the data may be used in either of the two execution units.

1.2.1. Loads and Stores

A LOAD instruction first computes the memory address. Next, it initiates a data transfer. The value from memory is then enqueued in the input FIFO in the appropriate execution unit. If multiple LOAD instructions are executed, the values are enqueued in the FIFO in the order of the LOAD instructions. The execution unit can access the enqueued data by referencing register 0 as a source operand.

A STORE instruction behaves somewhat differently. To write a value to memory, the value is written to register 0. Next, at some later time, a STORE instruction computes the memory address and initiates a data transfer. Alternately, a STORE instruction could first compute the memory address, and the transfer could be initiated at a later time when a value is written to register 0. As in the case of the LOAD instructions, multiple assignments to register 0 result in the values being queued in the FIFO, and when the memory addresses for these become available (through a STORE instruction), they are written to memory in the order in which they were enqueued.

The IEU does not block on a LOAD or a STORE instruction. If an execution unit (IEU/FEU) names register 0 as a source to access a memory operand, the FIFO is dequeued. If there are no values in the FIFO (i.e., the memory system has not yet enqueued the data), the execution unit blocks here waiting for the data. The same principle applies to the stream instructions.

1.2.2. Streaming

Another way by which the execution units can access memory is through a feature called "streaming." Streaming is a method of loading and storing structured data elements without having to do explicit address computations for each element. A stream instruction specifies a base address, a count, a stride, and which FIFO to use [8]. The general form of a stream instruction is:

S<in/out> FIFO, base, count, stride

Only one input and one output stream per FIFO may coexist. This implies a maximum of four simultaneous streams (two input, two output) per execution unit.

An example of a stream instruction is:

```
Sin32i r1, r5, 50, 16
```

This instruction causes 50 integers (32 bit values) with the base address equal to the value in register r5, at a stride of 16 bytes from each other, to be loaded in the IEU FIFO r1. It is important to note that, like the LOAD/STORE instructions, the values are enqueued in the FIFO in the correct order.

1.3. FIFO Depth Estimation

Since the IEU and the FEU go through FIFOs to access the memory, it is important to characterize the FIFOs, and understand their working. This report deals with estimating the depth of the FIFOs. Making FIFOs shallow implies that the memory system and/or the execution unit have to wait to enqueue or dequeue values, and making them too deep consumes more silicon area than necessary. This report tries to answer the question: how deep is "deep enough?"

2 Modus Operandi

On most architectures, the memory system is the performance bottleneck. To access an operand from memory the processor has to initiate a load and wait until the operand becomes available. The problem could be alleviated by loading the value much before it was needed, so that the slower memory system could cope up with the processor requirements.

On conventional machines, the number of prefetches possible is limited by the number of free registers. In addition, if the operands are being fetched in a loop (typically, array accesses), then prefetching into free registers is not a feasible option. If the operands are to be prefetched, the loop has to be unrolled, and instructions juggled around to use the free registers.

2.1. Why Use FIFOs?

The WM FIFOs are a convenient way to solve this problem. The prefetched values are simply loaded into the FIFOs, and are dequeued in the order in which they were loaded. Thus, FIFOs can be used to hide the slower memory system from the processor. They "average" the "bursty" requirements of the processor, in the sense that if the memory system can supply the operands at the average rate required by the loop, as opposed to its peak requirements, we won't stall the processor.

This can be illustrated with an example. Consider a loop that has eight instructions, and involves four memory accesses. The time chart for the loop is shown in Figure 2.1. Notice that the average memory bandwidth required is four operands in eight cycles, while the peak requirement is two operands per cycle. Figure 2.1-B shows a possible operand prefetch pattern.

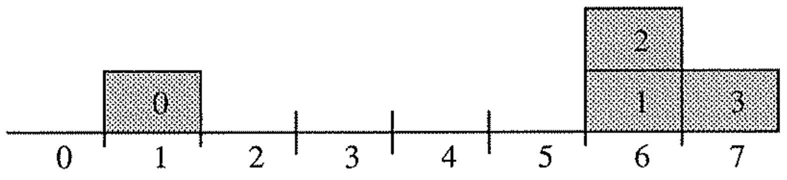
Thus, operand 0 is loaded in cycle 0, and operands 1, 2 and 3 in cycles 2, 4 and 6 respectively. This averages the bursty requirement, and the memory system need only supply operands at the rate of one operand every two cycles. As can be seen from Figure 2.1-C, a FIFO depth of three is sufficient for this example.

Let us now come back to the question posed in chapter 1: How deep is "deep enough?" In the above example, if a depth of two had been chosen, the memory system could not have enqueued operand 3 in the FIFO in cycle 6, because the FIFO would have been full. Operand 3 would have to be enqueued in cycle 7, where it was required by the processor. This would involve a wait by the processor for the operand to become available.

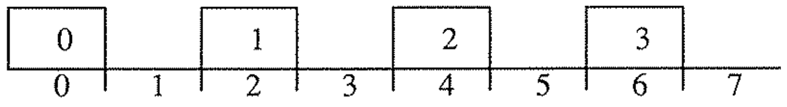
2.2. Methodology

To gauge the FIFO depth requirements, source code from some benchmarks and code for some other commonly used algorithms was compiled into the WM assembly language. For some of the benchmark routines, the source code had to be "tweaked" to take advantage of the WM architecture. A "smart" compiler could have done these optimizations, but in the absence of one, the changes were made manually.

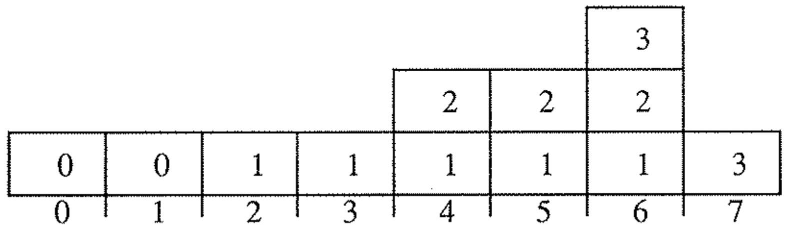
Once these routines were in WM assembly language, the inner loops were examined to see the FIFO access patterns. The reason for examining only the loops was that this was where the programs spent most of their time in execution. A time diagram was drawn for the loop instructions, and the FIFO depth required was computed from this diagram. This was repeated for all the routines studied, and the results tabulated.



A: Filled rectangles represent operands dequeued at that cycle



B: Rectangles represent operands enqueued at that cycle



C: Rectangles represent FIFO contents during that cycle

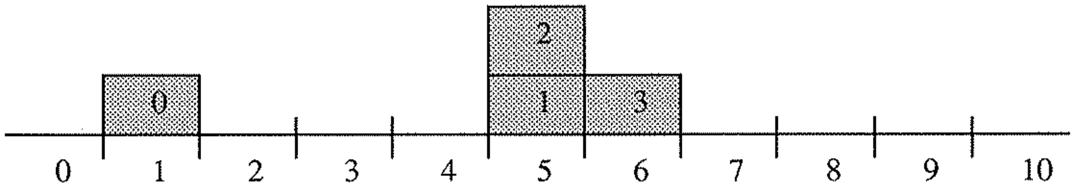
Figure 2.1

To illustrate the technique, let's consider an example:

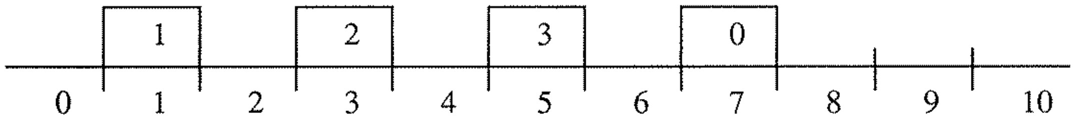
```
L1:
     LW      r31 := r26
             r20 := r0
     LW      r31 := r26 + 4
     LW      r31 := r24
     LW      r31 := r24 + r5
             r20 := (r0 + r20) + r0
             r20 := (r0 + r31) + r20
             r26 := r26 + 4
             r24 := r24 + r5
             r31 := r24 <= r3
             JumpIT L1
```

Example 2.2.1

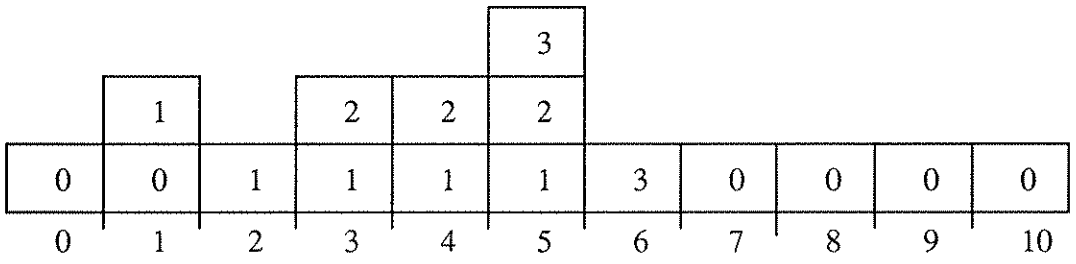
The time line for the instructions of this loop is shown in Figure 2.2.1. In part A of the Figure 2.2.1, the filled rectangles above the time line represent the operands required at that cycle. In part B, the rectangles represent a possible load pattern. Thus, operand 0 is available before the loop starts, and at cycle 0 of each loop iteration; operand 1 is enqueued during cycle 1, and so on. Part C depicts the contents of the FIFO as the loop is executed. From part C, it is clear that a FIFO depth of three is sufficient for this loop.



A: Filled rectangles represent operands dequeued at that cycle



B: Rectangles represent operands enqueued at that cycle



C: Rectangles represent FIFO contents during that cycle

Figure 2.2.1

It would seem that the FIFO depth is a function of the average rate of memory accesses, and the peak (burst) memory access rate. The next example shows that it is not a function of only these parameters:

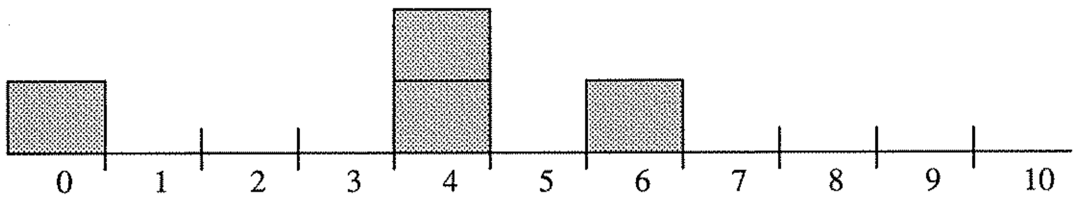
```
L2:
             r20 := r0
     LW      r31 := r26 + 4
     LW      r31 := r24
     LW      r31 := r24 + r5
             r20 := (r0 + r20) + r0
             r26 := r26 + 4
             r20 := r20 + r0
     LW      r31 := r26
             r24 := r24 + r5
             r31 := r24 <= r3
             JumpIT L2
```

Example 2.2.2

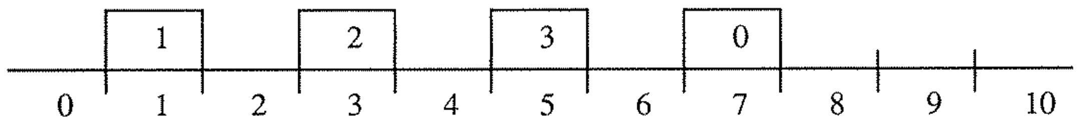
From Figure 2.2.2, it is clear that a FIFO depth of two is sufficient for this loop. Observe that for both the loops, the average number of accesses per cycle are the same (four operands in eleven cycles). In addition, both the loops place the same peak demand for memory accesses (two operands per cycle). It is clear, then, that the FIFO depth required depends not only on the peak and the average memory bandwidth, but also on the sequence of instructions in the loop.

2.3. Why does this method work?

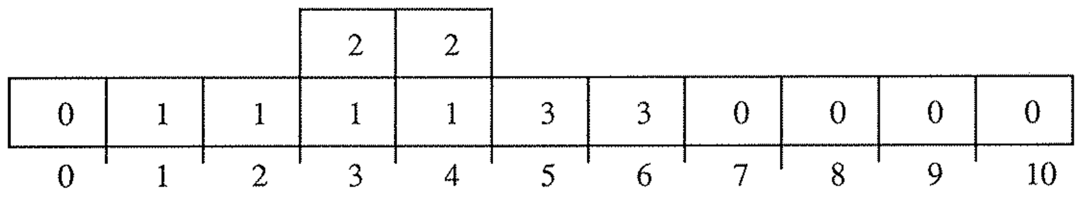
At this juncture, one might well ask: Don't the LOAD instructions limit the rate at which the instructions can be executed? After all, if an instruction that uses a memory operand (register 0) occurs immediately after the load instruction, wouldn't the execution



A: Filled rectangles represent operands dequeued at that cycle



B: Rectangles represent operands enqueued at that cycle



C: Rectangles represent FIFO contents during that cycle

Figure 2.2.2

unit have to wait until the operand is loaded, no matter what the depth of the FIFO? Also, why do we consider only the case where the memory system can supply data at the average rate required in the loop? To answer the first question, let us consider the implications of software pipelining and streaming on our examples.

2.3.1. Software Pipelining

Software Pipelining is a technique for reorganizing loops such that each iteration in the software-pipelined code is made from instruction sequences chosen from different iterations in the original code segment [1,3]. A software pipelined loop interleaves instructions from different iterations without unrolling the loop. There is also some start-up code that is needed before the loop can start, and some clean-up code to finish up after the loop terminates. Example 2.3.1 is a software-pipelined version of the loop in example 2.2.1.

The start-up code loads all the operands needed for the first iteration. The operands loaded in the first iteration of the loop will be used in the second iteration, and so on. For the WM architecture, software pipelining is possible with little or no changes to the loop body. This is because the loads enqueue data in the FIFOs in the order of the load instructions, so that a reference to register 0 always fetches the correct operand. The clean-up code uses the operands loaded in the last iteration of the loop. Since the software-pipelined version of the loop body is the same as the original non-software-pipelined version, we can think of the earlier examples as being software pipelined.

Software pipelining looks very similar to loop unrolling, but, as in the example 2.3.1, we do not unroll all the instructions in the loop. One advantage of software pipelining over loop unrolling is that software pipelining takes less code space. Software pipelining and loop unrolling have different objectives — loop unrolling reduces the overhead of the loop while software pipelining reduces the time when the loop is not running at peak speed to once per loop at the beginning and end.

```
     -- start-up code
     LW      r31 := r26
     LW      r31 := r26 + 4
     LW      r31 := r24
     LW      r31 := r24 + r5
             r26 := r26 + 4
             r24 := r24 + r5
             r3 := r3 - r5
     -- loop body, unchanged
L1:
     LW      r31 := r26
             r20 := r0
     LW      r31 := r26 + 4
     LW      r31 := r24
     LW      r31 := r24 + r5
             r20 := (r0 + r20) + r0
             r20 := (r0 + r31) + r20
             r26 := r26 + 4
             r24 := r24 + r5
             r31 := r24 <= r3
             JumpIT L1
     -- clean up code
             r20 := r0
             r20 := (r0 + r20) + r0
             r20 := (r0 + r31) + r20
```

Example 2.3.1

Thus, as evinced by example 2.3.1, software pipelining solves the problem of the IEU being blocked on a reference to r0, waiting for a LOAD to enqueue data. In the loop, the reference to r0 fetches data loaded in the previous iteration, which would be available if the memory system can deliver operands at the average rate required by the loop.

2.3.2. Streaming

Another way to keep the execution units from blocking on references to register 0 is to use streaming, described in Section 1.2.2. A stream instruction enqueues data in the designated FIFO, so that the loop body can remain essentially unchanged. There is, however, a restriction on what can be streamed in or out. Streaming requires that the elements be of a fixed size, and at a constant stride from each other. Streaming can be profitably used in applications that make a lot of regular array references, like the routines in the LINPACK suite.

2.3.3. Why Average Rate?

Let us consider all the possible cases — the memory system supplies data faster than the average rate required in the loop, slower than the required rate, and equal to the required rate.

When the memory supplies data at a slower rate than required, requests for operands will start piling up, and the memory system gets further and further behind the loop, as iterations continue. So, no matter how deep the FIFO, after a sufficient number of iterations, the execution unit *will* block on some instruction, waiting for data from memory. So, FIFO depth is not a crucial factor in this case. If the memory system supples data at the average rate, then, the execution unit will not block on an instruction if the FIFO is deep enough (Section 2.1). Lastly, if the memory system is faster, the operands are available at a rate faster than needed, and shallower FIFOs would suffice. In effect, then, as far as FIFO size is concerned, the “worst-case” happens when the memory system supplies data at the average rate required by the loop.

3 Results

We now tabulate the results of applying the technique described in the last chapter to some representative programs. These programs include the LINPACK routines, routines from the Stanford suite, some substring matching routines, the Poisson algorithm for solving partial differential equations, and the Simplex algorithm.

3.1. The LINPACK Routines

LINPACK was chosen because it models the code used in scientific algorithms very closely. The loops in LINPACK routines are characterized by small code size and long execution time. LINPACK uses a lot of operations on vectors and matrices, which lend themselves to streaming.

The results are tabulated in Table 3.1. It might come as a bit of a surprise to find that for most of the LINPACK routines, a FIFO depth of one is sufficient. The inner loops of most of these routines translate into loops with one or two instructions in the WM assembly language. This is because of two features of the WM architecture— streaming, and the fact that one WM instruction can perform two operations.

The routines `matgen` (generate a matrix), `daxpy` (scale a vector and add it to another) and `ddot` (dot product) have operations on two arrays, and use two FIFOs. All of them result in loops with one or two instructions, and are memory bandwidth limited (need more than one memory access per cycle, Table 3.1.)

As an example, the inner loop from `daxpy` is reproduced below:

```
L2:
        double f20 := f0
        double f0 := (f22 * f1) + f20
L1:
        JNIf0 L2
```

This loop results in two memory reads and one memory write for each iteration. The data read from memory is enqueued in FIFOs f0 and f1, and the data to be written to memory is enqueued in f0 (note that f0 names both an input and an output FIFO). The depth of the FIFOs would have little effect on the rate of execution — performance is limited by the memory bandwidth. Applying the technique described in chapter two, we note that a depth of one is sufficient for `daxpy`.

Routine	FIFO Depth	Memory Bandwidth (# accesses/cycle)
matgen	1	3
daxpy	1	1.5
ddot	1	2
dscal	1	2
idamax	1	2
dmxpy	4	2

Table 3.1: FIFO Depths for the LINPACK Routines

The procedure `dscal` (scale a vector) and the function `idamax` (return index of element with maximum absolute value) operate on one array, and consequently, use one FIFO. Both are memory bandwidth limited (two memoary accesses per cycle), and using an argument similar to the one used for daxpy, a FIFO of size one is sufficient for `dscal` and `idamax`.

`dmxpy` multiplies a matrix `m` with a vector `x`, and adds the result to another vector `y`. We need three input vectors (m, x and y), and one output vector, y. The code from one of the loops in `dmxpy` is reproduced below:

```
L135:
             r20 := (r22 - 1) ^ 3
     LD      r31 := (r28) + r20
     double  f20 := f1
     LD      r31 := r8
     double  f20 := (f0 * f0) + f20
     LD      r31 := (r22 ^ 3) + r28
     LD      r31 := r9
     double  f0 := (f0 * f0) + f20
             r8 := (r8) + 8
             r9 := (r9) + 8
L134:
     JNIf1 L135
```

One of the input vectors is streamed into f1, and the other two are loaded into f0 as their values are needed. The loop makes four memory accesses (three read, one write) in two cycles and is limited by the memory bandwidth. (Note that the floating pint instructions are executed in a separate execution unit.) Again, appealing to the technique described in chapter two, a FIFO of size four is sufficient for `dmxpy`.

3.2. The Stanford Suite

The Stanford Suite of benchmarks contains different kinds of routines. Some of them (Towers of Hanoi, n-Queens etc.) are recursive, while others (FFT, matrix multiplication etc.) show a propensity for array accesses and computation. Table 3.2 lists the FIFO depth required for the routines in the Stanford suite.

Routine family	FIFO Depth	Memory Bandwidth (# accesses/cycle)
Perm	*	-
Towers	*	-
Queens	4	.3
Mm	1	2
Puzzle	1	.4
Quick	1	.5
Trees	*	-
Fft	4	.3

Table 3.2: FIFO Depths for the Stanford Suite

The routines marked with an asterisk (`Perm`, `Towers`, `Trees`) are recursive, and do not make great demands on the memory system, and hence FIFOs do not affect performance appreciably.

`Queens` is recursive. However, one of the routines, `Try`, is very interesting. `Try` has a loose loop, and four instructions that write to the output FIFO r0. These instructions occur close together in the loop. Since writes to the memory are performed at the average rate required in the loop, these instructions succeed in enqueuing four values in the output FIFO r0 before they can be stored to memory. A FIFO of size four is, therefore, required for this loop.

Mm multiplies two real matrices, and uses Innerproduct which is very similar to ddot in linpack, and using the same line of reasoning, a FIFO of size one is sufficient for Mm.

Puzzle and Quick both have loops that access array elements. However, these loops have other instructions, which relax the memory bandwidth required for these loops. Table 3.2 lists the FIFO depth required for these procedures.

In Fft the loop is not very tight. However, as in Queens, memory references come bunched together in the loop. As a result, a deeper FIFO improves execution rate by allowing the memory system more latitude in enqueuing operands. If the memory system is capable of supplying data at the average rate required in the loop, a FIFO depth of four would suffice. If the memory system is slower, the loop becomes memory bandwidth limited. Experiments on the WM simulator [2] have shown that the difference in time to execute FFT between configurations with FIFO depths of three and FIFO depths of fifteen is relatively small (less than six percent if memory bandwidth is not a bottleneck).

3.3. Other Programs

The FIFO depth estimation procedure was also applied to some other programs. These were the substring matching algorithms (the Knuth-Morrison-Pratt algorithm, the Bayer-Moore search and the Rabin-Karp algorithm), the Poisson algorithm for partial differential equations, and the Simplex algorithm. The algorithms were derived from [5], and the results are tabulated in Table 3.3.

The loops in two of the substring match algorithms (Bayer-Moore, Knuth-Morrison-Pratt) are not very tight, and benefit from deeper FIFOs, as seen from Table 3.3. The Poisson algorithm uses two FIFOs. However, the FEU requires five memory operands in two cycles, and is severely restricted by the memory bandwidth. The Rabin-Karp and the

Program	FIFO Depth	Memory Bandwidth (# accesses/cycle)
KMP-Search	2	.4
Bayer-Moore	3	.3
Rabin-Karp	1	.1
Poisson	3	2.5
Simplex	1	.3

Table 3.3: FIFO Depths for some programs

Simplex algorithms are not memory bandwidth limited, and a FIFO of size one is sufficient for these algorithms.

4 FIFOs of Size Five or More?

The WM architecture mandates a depth of at least three for the input FIFOs. This is because an instruction with three operands may name register 0 (register 1) as source for all the operands. Thus, any instruction of the form:

r5 := (r0 **op1** r0) **op2** r0

would require that the depth of the FIFO r0 be at least three.

FIFOs of size four would suffice for the programs considered in the last chapter. However, before concluding that FIFOs of size four are good enough for most practical programs, one would like to speculate on the nature of programs that require FIFOs of size five or more. With this aim in mind, let us analyze the parameters that affect the depth of the FIFOs — the speed at which the memory system can deliver data, and the pattern in which instructions in the loop access the loaded data.

4.1. Memory System Constraints

To get a feel for the effect of memory speed on FIFO size, we consider the two memory bandwidth extremes — a memory system with "infinite" bandwidth, and a memory system with "near-zero" bandwidth.

For an infinitely fast memory system, a FIFO of size three would be sufficient for any application. In a particular cycle, an instruction can refer to at most three FIFO elements, and all these can be loaded just before the cycle starts. In effect, a FIFO of size three would be adequate for *any* program one could device.

Consider the other extreme, i.e., a very slow memory system. To avoid stalling the execution unit, all the data would have to be prefetched before the loop could begin execution. This is equivalent to unrolling the loop, and prefetching all the data. That makes the depth of the FIFO equal to the number of operands being loaded into that FIFO during the loop execution.

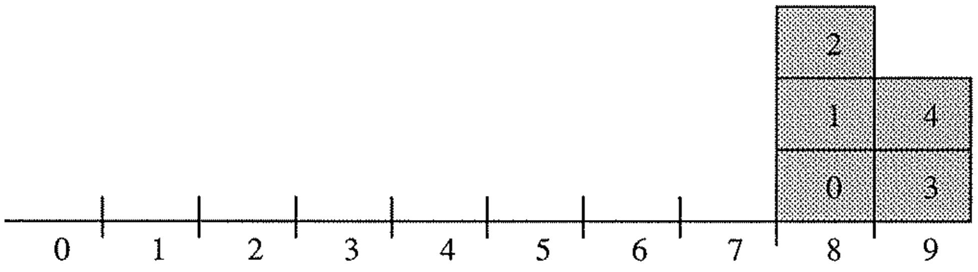
In the remainder of this chapter, as in the earlier chapters, we consider a more "average" memory system. In the discussions that follow, we assume that the memory system bandwidth is equal to the average bandwidth required by the loop.

4.2. Data Access Pattern

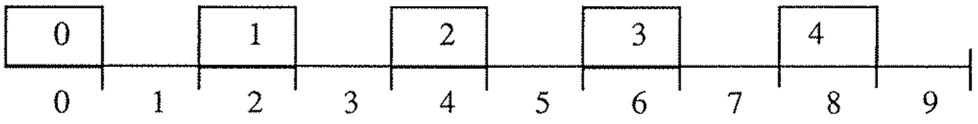
FIFO depth is affected by the pattern in which the FIFO contents are accessed. If the instructions that access the FIFO occur bunched together in the loop, a deeper FIFO would be required to mask memory latency. Conversely, if the instructions access the FIFO in a more even manner a smaller sized FIFO would suffice.

It is apparent that to require a FIFO of size five, the loop would have to contain at least five references to the FIFO. Figures 4.2.1 and 4.2.2 are two examples where a FIFO of size five is required. As described in chapter two, part A of the figure shows the FIFO reference (dequeue) pattern, part B shows a possible load pattern, assuming that the memory bandwidth is equal to the average memory bandwidth required in the loop, and part C shows the FIFO contents as a function of time. Figure 4.2.3 shows a loop where a FIFO of size two is sufficient.

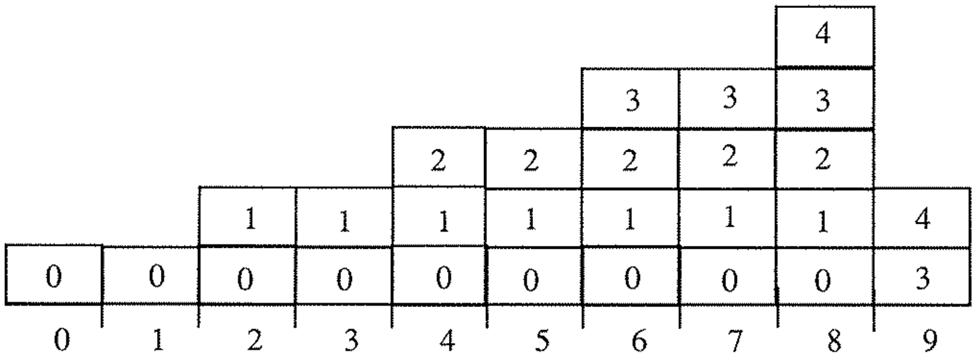
Note that all the three loops require the same average bandwidth (one memory access every two cycles.) What can one now say about the "spread" of the instructions that dequeue the FIFO? In all the three cases, the memory system enqueues data at periodic intervals. However, in the loops in Figure 4.2.1 and 4.2.2, since the instructions are



A: Filled rectangles represent operands dequeued at that cycle

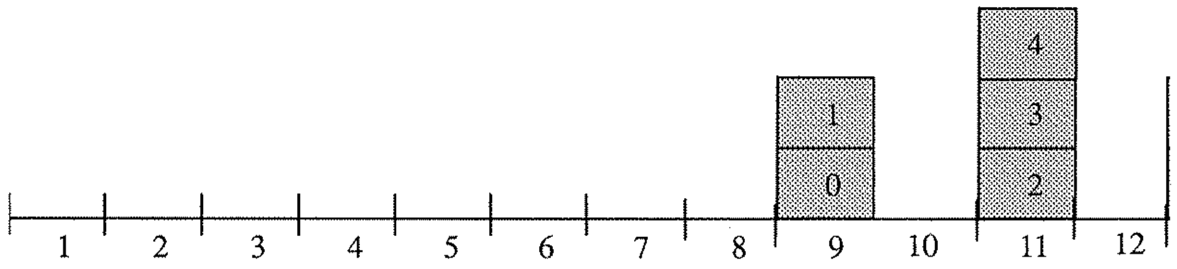


B: Rectangles represent operands enqueued at that cycle

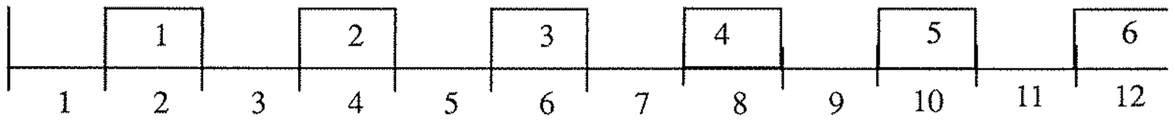


C: Rectangles represent FIFO contents during that cycle

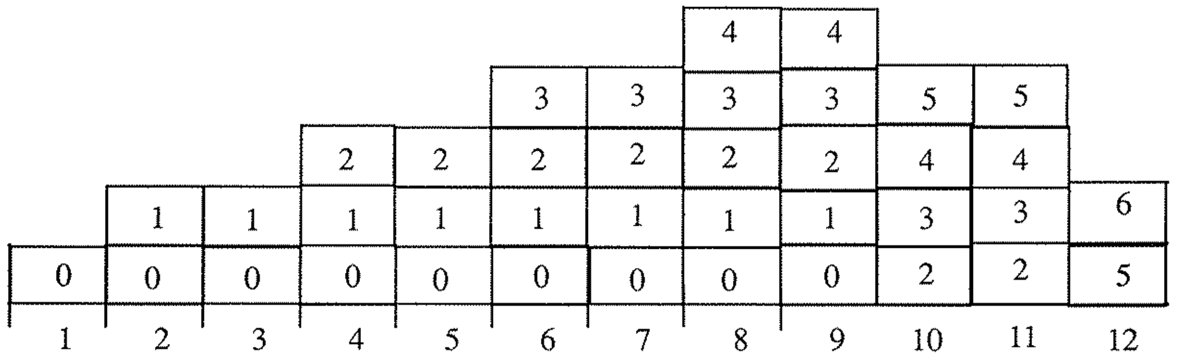
Figure 4.2.1



A: Filled rectangles represent operands dequeued at that cycle

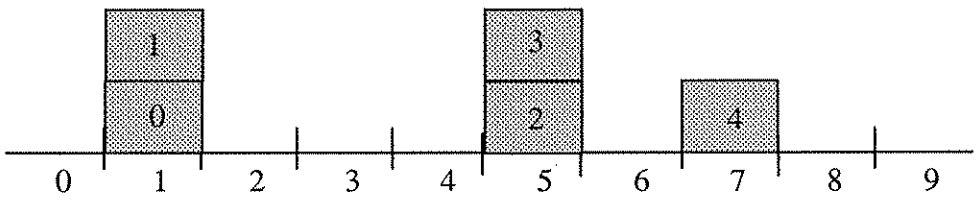


B: Rectangles represent operands enqueued at that cycle

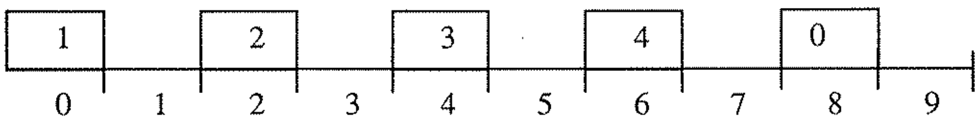


C: Rectangles represent FIFO contents during that cycle

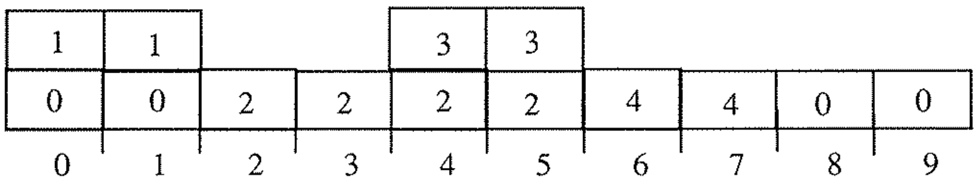
Figure 4.2.2



A: Filled rectangles represent operands dequeued at that cycle



B: Rectangles represent operands enqueued at that cycle



C: Rectangles represent FIFO contents during that cycle

Figure 4.2.3

"bunched" together, the memory system succeeds in enqueuing five values before the execution unit consumes (dequeues) these from the FIFO. On the other hand, in the loop of Figure 4.2.3, since the instructions that consume the FIFO appear at fairly regular intervals, the data enqueued by the memory system gets consumed by the execution unit before five values can accumulate in the FIFO.

4.3. FIFOs of Size Five

To summarize, loops that require a FIFO of depth five should have at least five references to the same FIFO. These references should be bunched, so that data values can accumulate in the FIFO before they are consumed. This "bunching" should be logically required by the program, in the sense that the compiler should not be able to shuffle instructions to insert some instructions between these bunched instructions. In addition, the total loop size should be such that the loop is not limited by memory bandwidth, but depends on the FIFO to even out memory requirements.

It would be relatively easy to find loops that satisfy these conditions individually, but finding programs that satisfy all the conditions in conjunction would prove to be difficult. This is because the loop would have to contain enough instructions to prevent the memory system from being a bottleneck, and at the same time have bunched references to a FIFO that cannot be separated by shuffling the instructions. None of the programs that have been considered satisfy these constraints.

5 Summary

Data FIFOs in the WM architecture improve performance by permitting memory delays to be overlapped with instruction execution. The depth of a FIFO depends on the average rate of memory accesses and the proximity of references to the FIFO. Code sequences in which FIFO references that occur bunched together require deeper FIFOs to mask memory latency, while those in which FIFO requests are more evenly spread out usually require smaller sized FIFOs.

FIFO size requirements for various programs were computed. It was observed that a FIFO of size four was sufficient for all the programs analyzed. Examples were then constructed to find the characteristics of programs that might need FIFOs of depth five or more. From these characteristics, it can be concluded that applications that require FIFOs of depth five or more without being limited by the memory bandwidth would occur quite infrequently.

References

[1] J. L. Hennessy and D.A. Patterson, "Computer Architecture A Quantitative Approach," Morgan Kaufmann Publishers Inc., 1990, pp. 325 - 328.

[2] P. J. Kester, "Performance of the WM Architecture: The Impact of Buffering Information Between Functional Units," Masters Thesis (Computer Science), University of Virginia, January 1991.

[3] M. Lam, "Software Pipelining: An Effective Scheduling Technique for VLIW Machines," Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation, June 1988, pp. 318-328.

[4] A. R. Pleszkun and E. S. Davidson, "A Structured Memory Access Architecture," in 1983 International Conference on Parallel Processing, August 1983, pp. 461-471.

[5] R. Sedgewick, "Algorithms," Addison-Wesley Publishing Company, 1988.

[6] J. E. Smith, "Decoupled Access/Execute Computer Architectures," Ninth Annual Symposium on Computer Architecture, April 1982, pp. 112-119.

[7] J. E. Smith, et. al., "The ZS-1 Central Processor," Computer Architecture News, volume 15, No. 5, October 1987, pp. 199-204.

[8] Wm. A. Wulf, "The WM Computer Architectures: Principles of Operation," Computer Science Report No. TR-90-02, January 1990.