A Methodology for Evaluating the Performance of RISC Processors

M.S. Obaidat
Dirar S. Abu-Saymeh

Computer Engineering Research Laboratory
Department of Electrical & Computer Engineering
University of Missouri - Columbia
Columbia, MO 65211

Abstract
A new methodology is devised to simulate and evaluate the performance of RISC processors. The detailed model provides a way to evaluate the performance of the processor for various applications and under a wide spectrum of operation environments and conditions. The instruction flow of each instruction is modeled. The instruction requests the various units and stages of pipelines which are modeled as resources. The model is applied on the Motorola MC88100 RISC processor which includes four processing units and numerous pipelines to speed up the execution of instructions. From the model the processing speed is investigated for various applications, and the contention on some units is studied. Moreover, the utilization of units and mean waiting time are studied and discussed.

I Introduction
Since the advent of the first computer in the forties, there has been a considerable effort to enhance the performance of computer machines. Many design directions have emerged. RISC (Reduced Instruction Set Computers) was one of these directions that appeared in the eighties.

RISC microprocessors feature the highest growth rates in the microprocessor sector due to their advantages. A typical RISC processor executes most instructions in a single cycle, has several general-purpose registers and large caches, can process several instructions simultaneously by pipelining and can fetch the next instruction while the current instruction is being executed. All of this was possible, because of the small instruction set and limited addressing modes which simplifies the design of the control unit. Thus leaving more space in the VLSI Integrated Circuit for other units to be stacked into the same chip. Moreover, the CPU can be designed using hardwired control units for faster execution. The RISC processor represents an attractive approach for GaAs, single chip, CPU realization [1,2,3].

The advantages of RISC processors for HLL and simple computations have been justified [1,2,4,5]. The simple instruction set and addressing modes makes the RISC an ideal processing unit for real-time applications.

The performance of a computer system can be evaluated by one of three methods: Direct measurement of the physical system; Analytical evaluation; And simulation. Previous work on RISC processors evaluation have focused on justifying the choice of a certain instruction set over the other and measuring the performance of the processor by a benchmark program. Most of the benchmark programs used for RISC evaluation were written in the C language. Among the benchmark programs developed by Patterson and Sequin [6] are character string search (E), Bit Set, Reset, Test (F), Linked List Insertion (H), Bit Matrix Transposition (K), Quicksort (I), Ackerman, Puzzle (subscript), Puzzle (pointer), Recursive QSOrt, SED (Unix-environment Batch Text Editor), and Towers of Hanoi (game program) [6,7,8,9]. Average relative code sizes for these C benchmark programs were measured for the VAX 11/780, 68000, Z8002, PDP 11/70 and BBN C/70 with reference to RISC I and RISC II. Moreover, the execution relative speed was measured for Reg-to-Reg. ADD operation [9]. Katevenis [10] has conducted
a study concerning the HLL performance of the RISC as compared to Assembly language and he found that RISC programmer has a higher incentive to program in HLL. The whetstone benchmark program [11] which was developed by the Central Computer Agency of the British Government was used to test numerical computing by executing a substantial amount of floating point arithmetic. It was found that RISC II C Compiler or the VAX Compiler, run on the RISC II at 12 MHz is twice as fast as the VAX C Compiler run on the VAX [9]. Patterson has tested the performance of LISP and PSL for the RISC and he found that the performance of RISC II is competitive to the VAX [10]. Heath [12] has applied the optimised C Compiler, running the same EDN benchmarks to the 68000, Z8002, and RISC I and found that the RISC I is faster than the other two processors by a factor of 2 to 4. Hitchcock and Sprunt [13] have found that the performance gains due to multiple register sets are independent of instruction set complexity.

Happel and Jayasumana [14] have conducted a simulation study on a RISC machine with two-level caches using an address trace taken from an experimental processor running a real-time telephone application. They found that the average access time is improved more readily by increasing the first-level cache size than by increasing the second-level size.

Such studies certainly seems to be impractical and non-cost effective, to evaluate the performance by direct measurements, on numerous prototypes. There are many factors involved in the efficiency of the simulation analysis. These include the level of details included in the model, the complexity of translating the physical model into simulation model and the accuracy of the results [15].

The analytical solution of a simple computer system can be achieved. However, in most cases the system is too complicated, and an analytical solution becomes difficult to find, if not impossible [4,16]. In particular the complicated interconnection of pipelines in a RISC processor makes this method inappropriate. The simulation technique is attractive due to the fact that it is usually more accurate than analytical technique, less expensive and faster than the measurement approach [16,17,18]. Moreover, the simulation method provides flexibility in the model implementation and allows further experimentation that could be impossible to implement in the analytic approach and very expensive and may be catastrophic in the measurement technique. Therefore, simulation technique is adopted in this work.

Since the control unit of a RISC processor takes only a small percentage of the chip area, many manufacturers tend to stack various processing units and pipelines into the chip. This generally complicates the simulation of such processors. Modeling the processor, in a multiprocessing environment, as a single server with certain parameters obviously produces inaccurate results [17,18]. Here we introduce a more detailed model that can be used in any environment in which a RISC processor is being used. In section II the general model is presented while section III presents the model for MC68100 as a case study. Section IV describes the usage of the model. The results and discussion is presented in section V. Section VI is the conclusion section.

II The Model

In this model, instructions are grouped according to their flow in internal units and pipelines. The instruction flow of each group is modeled, with some units being accessed by one or more groups.

All units and stages of pipelines are modeled as resources, with a FCFS queue. An example on these stages are the prefetch and decode stages in an instruction pipeline which are available in almost all RISC processors. Data busses, address busses and internal busses are also modeled as resources, since only one instruction can use them in each cycle. All instructions are queued to the system. Obviously all instructions will need to be prefetched into the processor. Therefore the whole benchmark program can be queued into the prefetch queue. After that, instructions will move into the model, requesting the required resources, operating on them and then proceeding to the next required stage or resource.

Since an instruction waiting for a resource stalls the pipeline, current stage resource is not released until the next stage resource becomes available to the instruction. This way, no instruction can leave the previous stage unless the next stage is free, thus stalling the pipeline. So we note that the service time of each stage is not constant and depends on the various stalls in the pipeline. These stalls could be due to the fact that the instruction may need other busy resources, other than the stage resource (i.e. busses, operands, etc.) and it could also be due to stalls of other instructions in the pipeline. The appearance of a branch instruction in the prefetch and decode pipeline in most processors would cause the pipeline to be cleared and directs the processor to fetch instructions from the new instruc-
tion location. However, the idea of delayed branching appeared as a solution to alleviate the waiting time to refill the pipeline. Delayed branching technique does not clear the pipeline. Instead the instructions currently in the pipeline are allowed to execute. Thus the programmer is encouraged to insert a useful instruction following the branch instruction which will be executed whether the branch is taken or not. If no such instruction is available the programmer should insert a no operation instruction. Advanced RISC processors give the user the choice of using delayed or non-delayed branching. The developed model takes care of both cases. If non-delayed branching is encountered the instruction pipeline is cleared.

Finally, we should model data dependencies, that is, if a previous instruction uses a certain operand as a destination for the result, then if the current instruction operates on that operand, it should wait until the previous instruction finishes execution. To model this, all operands are modeled as resources. If a processor operates only on registers, which is the case in almost all RISC processors, then only the registers are modeled as resources. But if the processor operates on memory, then the memory should be modeled as an array of resources, with the index of the array being the address of that memory location. Each instruction, before execution, should request the required source and destination operands. The source operands are released directly after reading their contents, while destination operands are only released when execution ends and results are obtained and stored in the registers. Thus, when the next instruction in line requests to operate on the operand being used in previous instructions, the request is denied and the instruction stalls the pipeline in which the instruction waits until the request is granted. An interesting feature of some RISC processors is the register windows. They can easily be modeled by using a register base counter which will provide the base for register access. When accessing registers, the register base counter is added to the register number to obtain the actual register number. The register base counter is incremented by a subroutine call and it is decremented by a return from the subroutine. The increment and decrement value depends on the RISC architecture.

The performance measures considered in this work are:

- Execution Time ($T_e$): which is the time needed to execute the whole benchmark program. This measure is given in clock cycles.
- Mean waiting or delay time ($T_w$): which is the mean number of cycles that instructions wait for their resource request to be granted.

Utilisation ($U$): which is defined as the ratio of time in which the unit is busy to the total execution time of the benchmark program.

$$U = \frac{\text{Total busy time}}{\text{Total execution time}}$$

III A Case Study: MC88100

The previous model was used to simulate the MC88100 RISC processor using SIMSCRIPT II.5 language [19]. Figure 1 shows a block diagram of the MC88100 processor [20]. The MC88100 includes four processing units: The Instruction Unit, Data Unit, Integer Unit and Floating Point Unit. The instruction unit performs the prefetching and decoding, and executes the branch instructions. The data unit executes the LOAD and STORE instructions. The integer unit executes the integer addition and subtraction and performs all logical instructions. The floating point unit executes all floating point instructions (fadd, fsub, fmul and fdiv) and integer division and multiplication. The floating point unit includes two pipelines for processing: the multiplication-pipeline, which executes both integer and floating point division and floating point addition and subtraction as shown in Figure 1. The two pipelines have a common first and final stages. An instruction flow diagram is developed and shown in Figure 2.

Each stage of the units and pipelines is modeled as a resource with FCFS queue. This provided 16 resources; 2 for instruction prefetch and decode, 2 for data bus address and reply, 1 for the data unit, 1 for the integer unit, 9 for the floating point unit and 1 for the write back to the registers.

The MC88100 has 32 registers. All arithmetic and logic instructions operate on these registers. In this case, we only need to model these registers as resources. Instructions request the required source and destination registers before they start execution. Source registers are released directly after reading their content with a write back, while destination registers are released after end of execution. This model takes care of data dependency.

Since current stage resource is not released until all required resources for the execution in the next stage are available, then the instruction stalls the pipeline. Note that, for example, if the instruction is in the first stage of the floating point unit, it will only stall the
Figure 1: Block diagram of the Motorola MC88100 RISC processor.

floating point pipeline. But if the next instruction in line is a floating point unit instruction, then that instruction cannot advance to the floating point unit and will stall the instruction pipeline.

IV Model Usage

This model is a detailed simulation of RISC processors. Thus the model can be used in any environment. In particular, we are interested in two cases, where the model gives accurate parameters. The first case is a RISC multiprocessor environment. In this case an ideal benchmark program can be submitted to the model. That will produce an exact and practical interaction between the RISC processor and the multiprocessor environment. The second case is a study of the performance of the RISC processor in general (in a uniprocessor environment). In this case we study the performance of the processor for different application programs. The instruction set is divided into groups according to their operation and flow in internal units. Any application can be characterised by the percentage of each group of instructions in the application program. For example, an application that involves data transfer would have a higher percentage of LOAD and STORE instructions. While a program that manipulates the data would have a high percentage of arithmetic and logic instructions. The performance is studied by varying the percentage of each group in a benchmark program. Due to locality of reference and the fact that programmers try to increase the data dependency distance (distance between instructions using the same operand), then the data dependency distances could be modeled as a poisson process with a certain mean. The relationships between the execution time of the benchmark, utilisation and mean delay time of different units versus the instruction group percentages are investigated and analysed for various mean data dependency distances.

V Results and Discussion

This section presents the results of performance evaluation prediction for the considered case study RISC chip. The instruction set is grouped into eight categories. These are:

- Group 1 which consists of the integer division instructions alone.
- Group 2 which consists of floating point division instructions alone.
Figure 2: Instruction flow model for the MC88100 RISC processor.
Group 3 which consists of the integer multiplication instructions alone.

Group 4 which consists of the floating point multiplication instructions alone.

Group 5 which consists of the floating point addition and subtraction instructions.

Group 6 which consists of the LOAD and STORE instructions.

Group 7 which consists of the integer addition and subtraction and logic instructions.

Group 8 which consists of the flow control instructions.

This classification is a result of the study of instruction flow in internal units and their operation.

Simulation experiments are performed to find out the value of performance measure versus benchmark program parameters. We use 3D plots to present the relations among various performance parameters and measures since this can give better insight to the understanding of the effect of different benchmark program categories on the performance index. In each experiment, two groups are chosen, and their percentage in a benchmark program is varied. The rest of the program was equally divided among the other groups. The instruction sequence is generated randomly to allow different combinations of consecutive instructions. The execution speed is a linear relation in all cases, with the division instructions heavily slowing down the execution. All other instructions are executing with a mean of 2 cycles, despite the mean dependency distance of 3. Figure 3 shows one of the execution time plots in which the percentage of floating point multiplication and division instructions in a benchmark program is varied. The benchmark program consists of 500 instructions. The plots nature is expected, since the division instructions need approximately 30 cycles to execute depending on whether it is single or double precision and integer or floating point division. The curves are used to predict the speed of execution of any program without running it.

The results on the write back resource seem to be more interesting. The mean waiting time for instructions to be able to write back to the registers is the highest when the program consists mainly of integer addition and subtraction, logic and LOAD-STORE instructions. Figure 4 shows one of these plots in which the percentages of integer multiplication, addition, subtraction, and logic instructions in a benchmark program are varied. The benchmark program consisted of 500 instructions. Note that the peak occurs at approximately 40% integer multiplication and 60% integer addition, subtraction and logic instructions. There is almost no delay time when the whole program is multiplication and a mean of .15 cycle when the whole program is integer addition, subtraction and logic instructions.

The utilization of the write back is deeply lowered by the increase of the division instructions in the program. Figure 5 shows one of the relations for the utilization of the write back to registers in which the percentages of floating point addition, subtraction and division in a benchmark program are varied. The figure shows the low utilization when the percentage of floating point division increases. Note that for 0% floating point division utilization is minimum when approximately 25% of the program instructions are floating point addition and subtraction.

The last stage of the floating point unit (FPLAST) could be a bottleneck since all floating point instructions join in that stage. The mean delay time for accessing that stage reaches its maximum when the program instructions are equally divided between integer multiplication and floating point addition and subtraction. Figure 6 shows the 3D relationship between mean FPLAST delay time versus the percentage of floating point addition and subtraction, and integer multiplication instructions. The other cases show a little effect on the delay time with floating point addition and subtraction and integer multiplication dominating the effect.

Utilisation of FPLAST is maximised when the whole program is integer multiplication, floating point addition and subtraction, or consists of a combination of both. The relationship between the FPLAST utilisation and the percentages of the instructions is shown in Figure 7.

VI Conclusions

To conclude, a new methodology for the evaluation of RISC processors is developed and presented. The model is applied to one case study, the Motorola MC68100 RISC processor. The model can be used in uniprocessor or multiprocessor environments when detailed modeling is needed for accurate results. Three performance evaluation measures are used, which in-
Figure 3: The execution time versus the percentage of floating point multiplication and division in a 500 instruction benchmark program.

Figure 4: Mean waiting time for instructions to be able to Write Back (WB) to the registers versus the percentage of integer addition, subtraction, logic and LOAD-STORE instructions in the benchmark program.
Figure 5: Utilisation of Write Back (WB) versus the percentages of floating point addition, subtraction and division in a 500 instruction benchmark program.

Figure 6: Mean waiting time for instructions to access the FPLAST stage versus the percentages of integer multiplication and floating point addition and subtraction instructions in a 500 instruction benchmark program.
clude utilisation of units, mean waiting time, and mean execution time. Other performance measures can easily be derived. It is found for the case study considered in this work that the average waiting time for the final stage of the floating point unit is maximum when the benchmark program is equally divided between floating point addition and subtraction, and integer multiplication. Utilisation of FPLAST stage is maximum when the application program consists of floating point addition and subtraction, and integer multiplication instructions. The mean execution time is minimum when the benchmark program consists of integer unit instructions. The utilisation of the write back is maximised when the benchmark consists of only floating point addition and subtraction. Finally, the mean waiting time for write back is maximised when the benchmark contains 40% integer multiplication and 60% integer addition and subtraction.

References


