Exploiting Register-usage for Saving Register-file Energy in Embedded Processors

Wann-Yun Shieh*, Chien-Chen Chen

Department of Computer Science and Information Engineering
Chang Gung University, Taiwan

259 Wen-Hwa 1st Road, Kwei-Shan Tao-Yuan, 333, Taiwan
TEL: 886-3-2118800-3336, FAX: 886-3-2118700
*Corresponding author: wyshieh@mail.cgu.edu.tw

Abstract. Low power register file design plays an important role in an embedded processor. In this paper, we exploit register-usage in a program to find out unused registers, and turn these unused registers into low power mode by annotating power-controlling instructions. The whole work is performed by applying the hardware/software co-design principle. For the hardware part, we propose a voltage-scaling control logic to supply voltages for each register. For the software part, we propose a power-controlling-code annotation approach to determine the voltage scaling behavior for each register. Simulation results show that the proposed approach outperforms the other related approaches in terms of the energy-delay product.

1 Introduction

In recent years, low power register file researches are lacking even though many other low-power hardware components, like BTB, ROB, TLB, etc, have been proposed. In general, the energy consumption of register file in an embedded processor is quite large. When we explore the sources of major energy consumption in register file, we find that all registers in an embedded-processor always keep active mode (1Volt) during program execution. This is because the registers must retain the values, which are produced from computing process. However, not all registers need to stay in active mode. The registers that need not to stay in active mode are called unused registers. Unused registers have two types. The first ones retain the values temporarily not to be used, and the second ones retain the values no longer to be used. These unused registers become the major sources of wasted energy consumption during program execution. Hence, finding out these unused registers in a program, and turning these unused registers into low power mode, i.e., drowsy mode (0.3Volt) or destroy mode (0 Volt), will result in large energy saving.

The objective of paper is to exploit register-usage in a program to find out unused registers for register file, and turn these unused registers into low power mode without large performance penalty. In order to achieve this objective, there are three important issues:

Issue 1: How to analyze register-usage in a program to find out unused registers?
Issue 2: How to control register voltages by different working modes?
Issue 3: How to limit energy and performance penalty?

The rest of this paper is organized as follows. Section 2 introduces and discusses related works, such as the dynamic voltage scaling technology and the low-power register file design. Section 3 proposes hardware/software co-design approach to exploit register-usage in a program for register energy saving. Section 4 discusses evaluation environment and evaluation results, including the metrics of register file energy consumption, processor energy consumption, performance penalty and energy-delay product. Section 5 gives conclusion and future works.

2 Related works

In this section, we discuss the development of the dynamic voltage scaling (DVS) technique, which is a well-known approach in reducing hardware energy consumption. Then we discuss some previous research about low-power register file design.

2.1 Dynamic Voltage Scaling Technique

The dynamic voltage scaling (DVS) technique is widely used to reduce power consumption or leakage energy for hardware components. The main idea of the DVS technique is to dynamically control supply voltages to a hardware by its working modes, including the active mode, the drowsy mode and the destroy mode, based on the access behaviors.

Most previous researches about the DVS technique focus on hardware solutions by utilizing hardware monitor to dynamically control the supply voltages. However, pure hardware solution has its drawbacks. For example, it may increase more hardware area due to the need of extra logic to control voltage scaling. In addition, the hardware solution does not precisely analyze overall program execution behavior, e.g., register usage; as a result, it cannot get much energy saving.

Due to these reasons, in recent years, the trend of the DVS technique has focused on coupling with the hardware/software co-design principle. This kind of approaches uses software analysis to get sufficient information from user programs, and then uses these information to help hardware components to switch among different working modes to get energy saving.

2.2 Hardware/Software Co-design for Register File

In [1], authors used the hardware/software co-design approach with the DVS technique to reduce register file energy. They analyze loop sections in a program to find unused registers, and then inserted power-controlling instructions before the loops to turn unused registers into drowsy mode. The advantages of this approach are easy analysis and simple implementation, because it focuses only on the loop sections. Also, it can reduce power-controlling instruction annotations, which may be the potential performance penalty. However, this approach, which did not consider other non-loop sections in a program, may possibly get little energy saving for the whole program.

3 Hardware/Software Co-design Approach

In this study, we propose another hardware/software co-design approach to exploit register usage for register-file energy saving. The proposed approach can be partitioned into the hardware part and the software part.

In the hardware part, we propose a voltage-scaling mechanism that contains two major components: the connected power-state-register design and the voltage-scaling control logic. The connected power-state-register is to record the power states (i.e., working modes) for multiple registers. The voltage-scaling control logic is to control different supply voltages to registers by individual working modes.

In the software part, we propose a power-controlling code annotation method that has two major components, including the power-controlling instruction design and the power-controlling code generation. The power-controlling instruction is to write a value into a power-state-register to change the working modes for multiple registers. The power-controlling code generation is to analyze the register-usage in a program and annotate power-controlling instructions into the program, based on a proposed voltage-scaling strategy, to determine the register working modes.

3.1 The Hardware Part：Voltage-scaling Mechanism

Fig. 1 shows the voltage-scaling mechanism. When a power-controlling instruction writes a new value into a connected power-state-register, the voltage-scaling control logic will switch new supply voltage to corresponding registers.

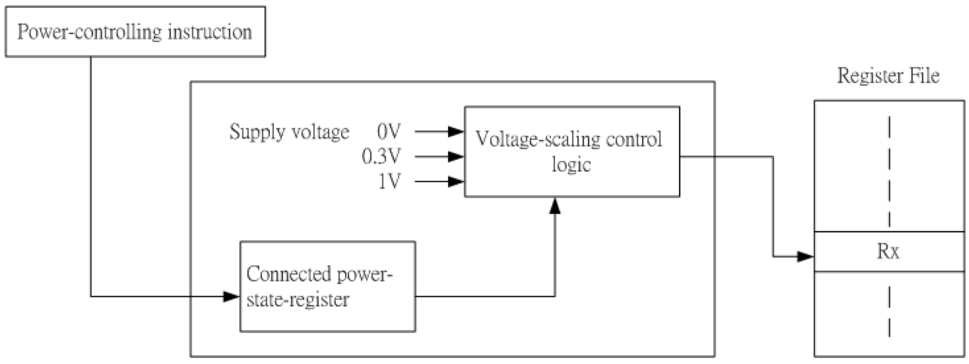

Power-controlling instruction
Register File
Supply voltage
0V
0.3V
1V
Voltage-scaling control logic
Connected power-state-register
Rx

Fig. 1. Voltage-scaling mechanism

3.1.1 Connected Power-State-Register Design

The design goal of the proposed power-state-register is to reduce power-controlling instruction annotations for saving potential performance penalty. In conventional design, each register is controlled by an individual 1- or 2-bit power-state-register, causing that multiple sequential power-state-register updates require multiple sequential power-controlling instruction annotations. Therefore we "connected" those individual power-state-registers into a single multi-bit register, called the connected power-state-register, for merging multiple sequential updates into just one annotation. The number of the connected power-state-registers for a register file can be determined by $\lceil 2n/m \rceil$, where n is the total register number to be controlled, and m is the default length of a connected power-state-register.

3.1.2 Voltage-scaling Control logic

The design goal of the voltage-scaling control logic is to control different supply voltages by different working modes, including the active mode (1Volt), the drowsy mode (0.3Volt), and the destroy mode (0Volt). In [1], authors proposed a control logic by applying the "switch" characteristic of transistors to turning on/off the supply voltages for each register. In this paper, we use the same control logic, but the power-state-register is replaced by the proposed connected power-state-register, as shown in Fig. 1.

3.2 The Software Part: Power-controlling Code Annotation

3.2.1 Power-controlling Instruction Design

The power-controlling instruction writes a value into the connected power-state-register to control working modes for multiple registers. Through the connected power-state-register design, each value-update (i.e., instruction annotation) requires only few ISA modifications. We can directly use the instructions, e.g., ADD or SUB, to change the value of a connected-power-state-register at the time that the working modes of some registers should change. These instructions will be annotated in the program execution code to control the working modes for each register.

Fig. 2 shows two examples to change the working modes of registers R0 and R1 through ADD and SUB instructions, respectively. In example 1, if we want to change the working mode of R0 from the active mode to the drowsy mode, we can update the value of the connected power-state-register (CPSRi) by annotating the instruction "ADD CPSRi, CPSRi, 1". This is because the bit-0 and bit-1 of the CPSRi control the supply voltages to R0, and state "01" represents 0.3-Volt drowsy mode. Similarly, we can annotate the instruction "ADD CPSRi, CPSRi, 3" to represent the change of working mode from the active mode (00) to the destroy mode (11). Example 2, on the other hand, shows how to change the working mode of R1 (i.e., bit-2 and bit-3) from the drowsy mode (01), or from the destroy mode (11) to the active mode (00).

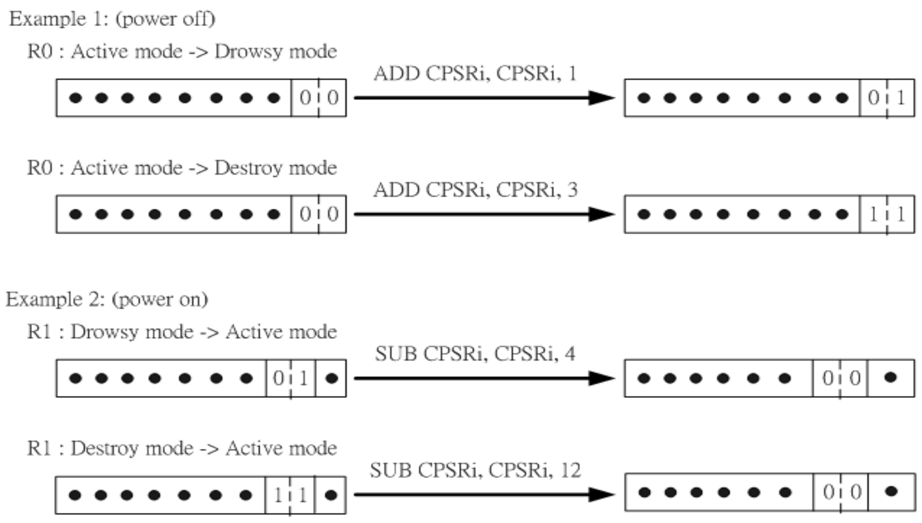


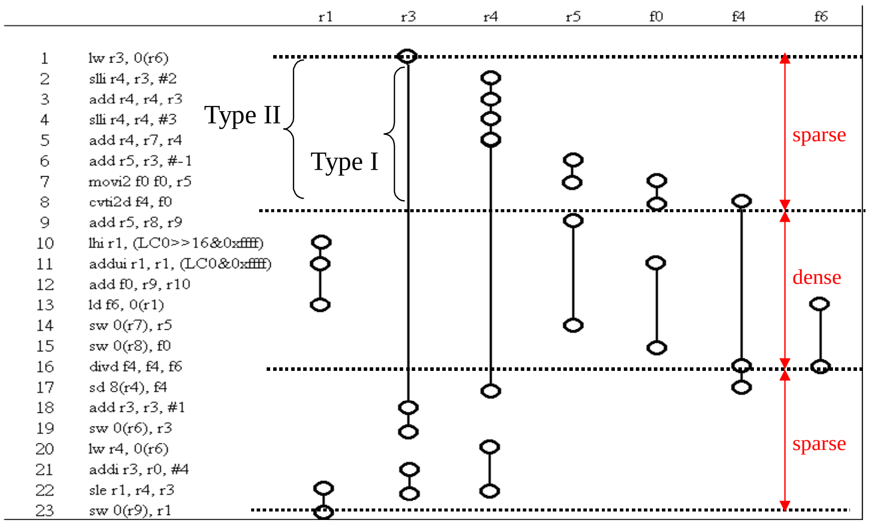
Fig. 2. Two examples of changing the working modes by ADD or SUB instructions

3.2.2 Power-controlling Code Generation

To precisely predict register-usages in a program, we take the trace file for analysis. The reason to do so is that we can exactly measure how the power-controlling code affects power consumption and performance penalty during program execution. First, the program is partitioned into sections for register-usage analysis. In [1] authors take the loop body as analyzed sections (we call it the Loop-based approach for convenience). In this paper, we propose another two more straightforward partitioning approaches, i.e., partition the program by fixed interval (called the FI-based approach), and by basic blocks (called the BB-based approach), for sections not limited to loops. Though these three approaches (FI-, BB-, and Loop-based) are employed for simple analysis and easy implementation, they cannot precisely reflect register-usage. A more "making-sense" option would be the approach that partitions the program by register-live-ranges (we call it the RLR approach).

Fig. 3 shows the register-live-range graph for a piece of instruction sequences in program basicmath of benchmark MiBench [3]. (In Fig. 3, we draw the live-ranges by solid lines, as well as the access point of register contents by small circles.) From the register-live-range graph, we found that the distribution of live-ranges is not so uniform; there did exist "sparse" and "dense" sections. The sparse section, in fact, represents the section that has more chances we can get energy saving from it, due to large amounts of unused registers. On the other hand, the dense section represents the section that the register pressure is heavy, and it is not worth to annotate extra power-controlling instructions to enlarge either energy or performance penalties. Therefore, in the RLR approach, only sparse sections will be taken for register-usage analysis. The partitions of the sparse sections and the dense sections can be performed by setting a scanning line across the register-live-ranges. When the scanning line scans along each instruction, if intersection numbers is less than a threshold (IT), we call it in a sparse section, and vice versa. After scanning the whole program, we get the sparse and dense sections alternatively appeared in the register-live-range graph.

In a sparse section, there are two types of ranges can be considered for energy saving. The first type, called the type-I range as shown in Fig. 3, means that the corresponding register is temporarily unused, and we may change it to the drowsy mode (the data will be kept for later use). The other type, called the type-II range, means that the value in the corresponding register is no longer to be used, and we can change it to the destroy mode. For both types of ranges, we will apply the same energy cost function to judge whether it is worth to insert power-controlling instructions.



```
1   lw r3, 0(r6)
2   slli r4, r3, #2
3   add r4, r4, r3
4   slli r4, r4, #3
5   add r4, r7, r4
6   add r5, r3, #-1
7   movi2 f0 f0, r5
8   cvti2d f4, f0
9   add r5, r8, r9
10  lhi r1, (LC0>>16&0xffff)
11  addui r1, r1, (LC0&0xffff)
12  add f0, r9, r10
13  ld f6, 0(r1)
14  sw 0(r7), r5
15  sw 0(r8), f0
16  divd f4, f4, f6
17  sd 8(r4), f4
18  add r3, r3, #1
19  sw 0(r6), r3
20  lw r4, 0(r6)
21  addi r3, r0, #4
22  sle r1, r4, r3
23  sw 0(r9), r1
```

Fig. 3. Example of register-live-ranges. The solid lines represent the live ranges, and a small circle represents the access point for a value

The RLR approach performs well balance between energy-consumption saving and performance penalty. First, by focusing only on the major sources of energy-consumption in sparse sections (i.e., type-I and type-II ranges), we can earn larger energy saving, but suffer less performance penalty. It is because only the sparse sections require power-controlling instruction annotations, not the whole file. Second, varying the IT threshold makes the size of sparse sections change differently; smaller IT results in more small-size sparse sections (and vice versa). These small-size sparse sections may not pass the energy cost function evaluation, and result in few instruction annotations. Thus, they earn less energy saving, but also suffer less performance penalty. (On the other hand, larger IT derives the opposite results.) The choice of IT, in fact, can be evaluated by experiments. Finally, by applying energy cost function evaluation to each instruction annotation, we can guarantee that any energy saving for register file would not cause extra energy consumption in pipeline; that is, minimize the side-effects for the whole processor energy.

We apply an energy cost function to each partitioned section to judge whether it is worth to annotate power-controlling instructions. Though the registers can achieve energy savings through working mode changes, the power-controlling instructions annotated may cause extra energy consumption in pipeline. Therefore, an energy cost function is used here to measure if energy saving of changing register working-mode is greater than the energy penalty of annotating extra instructions. If yes, then a voltage-scaling strategy will be activated to determine which working mode should a register change to. If no, nothing happens in this section, and the next partitioned section will be taken for analysis through the same processes.

4. Evaluation

In this section, we first present the evaluation configuration, including environment, parameters, and evaluation flow. Then we show evaluation results, in the terms of the register-file energy-consumption, the processor energy-consumption, the performance penalty, and the energy-delay product. In addition, we show how to select suitable IT threshold for the RLR approach.

4.1 Evaluation Environment

Table 1 shows evaluation environment, including the target processor, benchmark, and approaches for comparison.

First, we use ARMulator to produce the trace files of benchmarks. Then the trace files are fed into the power-controlling-code annotation process. In this process, we would evaluate four approaches including FI, BB, Loop, and RLR –based approaches by energy and performance analysis.

Table 1 Evaluation environment.

Simulator	Simplescalar 4.0 [9]
Energy model	Wattch 1.2 [4]
Target processor	ARM with 5 pipeline stages [10]
Benchmark	MiBench [3]
Approaches for comparison	FI, BB, Loop, RLR

4.2 Evaluated Results

A. Flexibility of the RLR approach – the effect of the *IT* threshold

The IT threshold plays the role of balancing the trade-off between energy saving and performance penalty. Different IT values would result in different influences. Applying a large IT in the RLR approach makes more large-sparse-sections; that is, suitable for energy-oriented applications. On the other hand, applying a small IT in the RLR approach makes less sparse-sections; that is, suitable for performance-oriented applications. Fig. 4 shows that, for example, when we set the value of IT to

8, we get larger energy saving and less performance penalty. We can derive the similar results from the experiments for other benchmark programs in MiBench.

B. Register File Energy Consumption

Fig. 5 shows the register-file energy-consumption for the four approaches. In the FI-based approach, we set the fixed-interval (i.e., *n*) by average length of basic blocks. In the BB-based approach, we partition the programs by basic blocks. In the loop-based approach, we partition the programs by loops. And, in the RLR-based approach, we partition the programs by register-live- ranges, and set the value of IT to eight.

Fig. 5 shows that for register-file energy-consumption, the RLR approach averagely outperforms the FI, BB, and Loop approaches by 21%, 18%, and 28%, respectively. These improvements come from that the RLR approach considers not only the type-I "temporarily-unused" register-live-ranges, but also the type-II "no-longer-used" register-live-ranges, and the type-II situations can be proved as the major wasted energy sources.

C. Performance Penalty

Fig. 6 shows the performance penalty for the four approaches. In Fig. 6, on average, the loop-based approach outperforms the other approaches by 17% ~ 23%. This is because the loop-based approach considers only the loop-sections in a program; hence, it results in the fewest power-controlling code annotations. On the other hand, the RLR approach performs better than the FI and BB -based approaches. This is because it focuses only on the sparse sections instead of the whole program. Hence, for the RLR approach, the number of power-controlling-code annotations can be limited.

D. Energy-delay Product

The energy-delay product is a widely-used metric to balance benefits in energy saving with any potential degradation in performance [2]. Fig. 7 shows the energy-delay products for the four approaches, and each approach is normalized to the FI approach. In Fig. 7, we find that the RLR approach would be a good choice among the others when the value of *IT* equals to 8. (It outperforms the other approaches by 21% - 67%.) This shows that the RLR approach gets a great balance between energy saving and performance penalty.

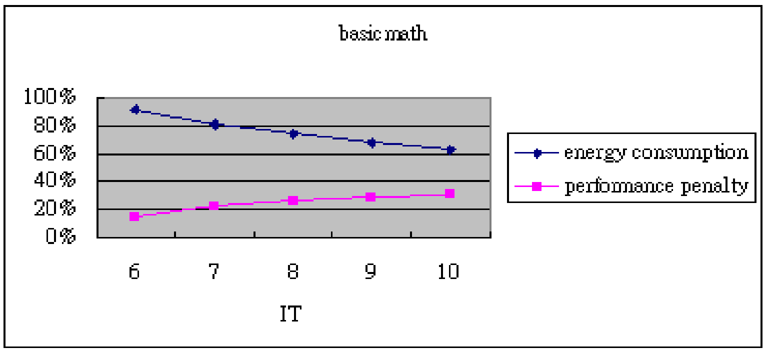


Fig. 4. Effect of IT in the RLR approach for basicmath of MiBench

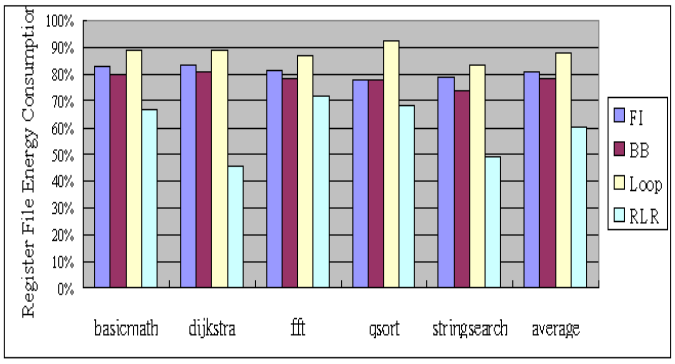


Fig. 5. Register-file energy-consumption for the four approaches

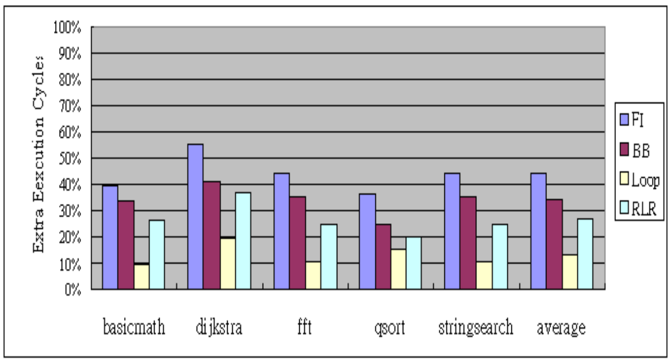


Fig. 6. Performance penalty for the four approaches

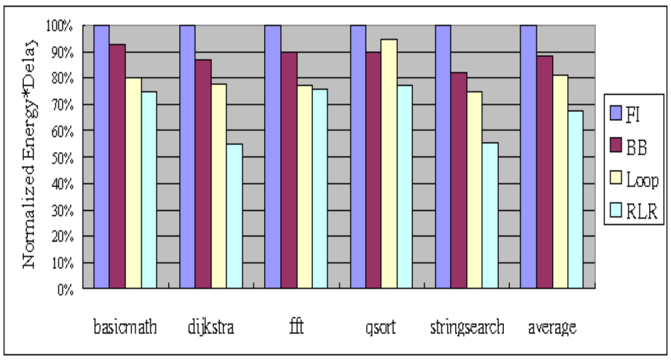


Fig. 7. Normalized energy-delay product for the four approaches

5. Conclusion

In this paper, we propose a hardware/software co-design approach for register-file energy saving. On the hardware part, we propose a voltage-scaling mechanism, including the connected power-state-register design and the voltage-scaling control logic. On the software part, we proposed a power-controlling code annotation mechanism, including the power-controlling instruction design and the power-controlling code generation. We partition the programs into sections to analyze register-usage. Simulation results show that the RLR-based approach outperforms the other partitioning approaches in terms of the energy, and energy-delay product. In addition, by varying the *IT* threshold in the RLR-based approach, the trade-off between energy saving and performance penalty can be well balanced.

There are some future works in this paper. For example, energy and performance penalty due to power-controlling code annotations could be reduced further by hardware supports. We know that a "power-on" instruction is annotated before a register value tends to be read or updated. In fact, we can catch this register-usage information in early pipeline stages, e.g., ID stage, without compiler notification. By forwarding a signal from the ID stage to a voltage-scaling control logic, we can power up a register on time for later use. Detailed design has been under development.

References

1. J. L. Ayala,et al.: Power-Aware Compilation for Register File Energy Reduction. International Journal of Parallel Programming, Vol. 31, No. 6 (2003)
2. W.Zhang, et al.: Reducing Instruction Cache Energy Consumption Using a Compiler-based Strategy. ACM Transactions on Architecture and Code Optimization, Vol.1, No. 1 (2004)
3. M. R. Guthaus, et al.: MiBench: A free, commercially representative embedded benchmark suite. IEEE International Workshop on WWC-4, (2001) 3 – 14
4. D. Brooks, et al.: Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. Proceedings of the 27th International Symposium on Computer Architecture (2000) 83 – 94
5. J. L. Ayala, et al.: Reducing Register File Energy Consumption using Compiler Support. Spanlish Ministry of Science and Technology under contract TIC 2000-0583-C02-02.
6. J. L. Ayala, et al.: Power-aware Register Renaming in High-Performance Processors using Compiler Support. Spanlish Ministry of Science and Technology under contract TIC 2003-0708.
7. J. L. Ayala, et al.: Energy-Efficient Register Renaming in High-Performance Processors. Spanlish Ministry of Science and Technology under contract TIC 2000-0583-C02-02.
8. J. Abella :On Reducing Register Pressure and Energy in Multiple-Banked Register Files. Proceedings of 21st International Conference on Computer Design (2003) 14 – 20
9. www.simplescalar.com/docs/simple_tutorial_v4.pdf
10. Http://www.arm.com/documentation/Software_Development_Tools/index.html