International Journal of Software Engineering and Knowledge Engineering

HW/SW CO-DESIGN FOR LOW POWER ARITHMETIC AND LOGIC UNITS

TAY TENG TIOW
Digital Systems and Applications Lab. National Univ. of Singapore, 10 Kent Ridge Crescent, Singapore 119260
eletttay@nus.edu.sg

NG KAR SIN
Digital Systems and Applications Lab. National Univ. of Singapore, 10 Kent Ridge Crescent, Singapore 119260
kar-sin.ng@hp.com

PAN YAN
Digital Systems and Applications Lab. National Univ. of Singapore, 10 Kent Ridge Crescent, Singapore 119260
panyan@nus.edu.sg

As many embedded microprocessors are battery driven, low power design is becoming increasing necessary. In this paper, we proposed hardware-software co-design architecture for low power arithmetic and logic units. By including multiple functional units with same functions and different speeds, we provide instruction implementations at various power prices. Then, with an assembler level scheduler, we identify and create situations whereby the low-power slow functional units can be utilized. The overall performance is not compromised as no additional stalls are introduced. Simulations show 10%~35% power saving in typical addition operations.

Keywords: Arithmetic Logic Unit design; Instruction scheduling; Instruction interdependence

1. Introduction

The booming of portable devices like PDAs and hand-phones owes much to the rapidly improving performance of the embedded microprocessors inside. To fit into these battery-driven devices, microprocessors are expected not only to execute complex functions but also to consume moderate power. Thus, low power processor design has become a hot research area.

It is a matter of fact that the power consumption of a microprocessor is proportional to the level of its performance. However, maximum performance is not always necessary for most applications. Thus, power saving can be achieved by cleverly lowering the performance level whenever feasible, leading to reduced overall energy consumption.

There are two aspects to the problem. First, hardware architectures should be designed to allow real time switching between different power-performance modes. Second, a software scheduler, representing the intelligence in the switching, should be designed to make the switching decisions.

A widely used technique that falls into this scheme is Dynamic Voltage Scaling (DVS) [1], whereby the working frequency of the microprocessor is lowered together with the supply voltage. The decision of when to scale the supply voltage is made by at least two approaches: by additional hardware monitoring certain execution statistics [2] or by code-based offline analysis [3]. Such designs are already found in commercial products [4].

However, the power saving achieved by these available techniques are still less than satisfactory. Some more aggressive methods should be taken to cut down more on the power budget. Our design is a full hardware/software co-design. Using a novel pipelined

ALU architecture, we provide a knob in the core of a microprocessor for adjusting power and performance. Together with this, we also develop an instruction re-shuffle algorithm to squeeze the most out of the slack due to inter-dependence between instructions.

The remaining sections of this paper are organized as follows. In sec. 2, we describe in detail the HW/SW co-design of our system. Simulation results are discussed in Sec. 3. Sec. 4 concludes the paper.

2. System Design

Our design focused on the ALU inside general purpose processors. The motivations are deduced from the following:

- Through simulation, we find that the energy consumption in fast Functional Units (FUs) is much higher than that in slower counterparts.
- There are many situations where fast FUs are not required.

To exploit the observation, we design an ALU where each arithmetic function may be implemented by many possible FUs, each with a different power or speed ratings. Then depending on the dependency between adjacent instructions, we optimally assign instructions to slower lower-power FUs as long as the performance is NOT compromised. A simple heuristic is employed in the offline assembler level scheduler to reshuffle the instructions so that instruction interdependence are lessened and further power savings can be achieved.

2.1. *Hardware Design*

2.1.1. *Power statistics of functional units of different speeds*

Similar to many other designs, the power saving is virtually always achieved at the price of performance. Through simulation, we found that the power consumption of fast FUs (*i.e.* adders, multipliers, *etc.*) is much higher than that of their slower counterparts. The following table shows the per-execution energy and data arrivals for fast and slow FUs. The simulation was carried out in Synopsis under .35um technology.

Table 1. Per-execution Energy and Data Arrivals for Fast and Slow Functional Units.

Functions		Fast	Slow	Difference (percentage)
Addition	Energy (pJ)	56	23	33 (58.9%)
	Data Arrival(ns)	3.77	12.00	8.23
Subtraction	Energy (pJ)	57	24	33 (57.9%)
	Data Arrival(ns)	4.13	12.51	8.38
Multiplication	Energy (pJ)	703	394	309 (44.0%)
	Data Arrival(ns)	8.17	27.11	18.94
Division	Energy (pJ)	1218	1049	169 (13.9%)
	Data Arrival(ns)	30.26	54.68	24.42

It is clear from the table that slower FUs consume considerably less energy. This is mostly because those fast FUs employs more logic circuits. Thus the internal parasitic capacitance and the number of signal transitions are much larger than the slower counterparts.

2.1.2. *Pipeline Structure*

In our design we have a pipeline structure as shown in Fig. 1. It is a generic five stage pipeline, but is sufficient to illustrate the strategy we employ. Our focus is on the execution stage and the write-back stage.

- Execution Stage. Here, we use multiple FUs of same functions. All the FUs are put in parallel and controlled by the Control Unit. At each clock cycle, at most one instruction is fed. Thus, we group all the FUs of same cycle times together and they will share a Common Output Register. The single issuing mechanism ensured that there will not be any conflicts in each group. As the FU cycle times vary among groups, instruction completion will be out-of-order and multiple instructions may complete together. Hazards are avoided by software control at the assembler level.
- Write-back Stage. As multiple instructions may complete at the same time, the register file has multiple In-ports to support multiple write-backs [5].

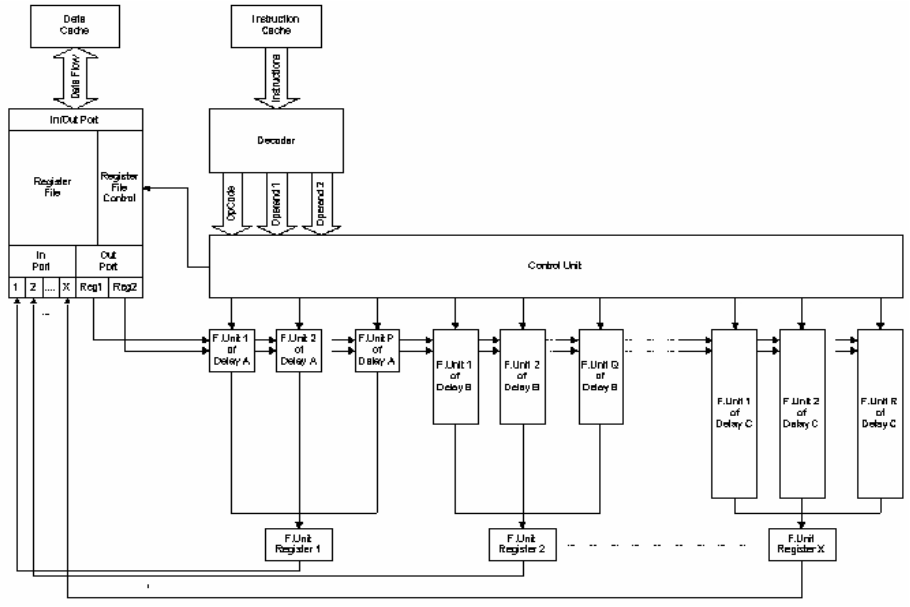

Fig. 1. Arithmetic Logic Unit Pipeline Structure.

With such a pipeline structure, we have multiple instructions performing a same function but with different power prices. The assembler level scheduler will then analyze the code and identify or create situations where arithmetic instructions can be executed with lower-power version FUs.

2.2. *Software Design*

An offline assembler level scheduler is designed to determine when to make use of the lower-power FUs instead of the default ones. The choice of different FUs is done by associating each FU with a different machine code. Here, the scheduler will intelligently map a particular program instruction (PIn) to possibly different machine codes, depending on the situation. Slower FUs will be preferred as long as it does not cause additional stalls.

The aim of the scheduler is not only to identify the situations whereby slower FUs can be used but also to create such situations by reshuffling instructions to resolve interdependence with a best effort. The processing of the original program codes is done in 2 phases.

2.2.1. *Initialization*

The first phase is to prepare the original codes and to convert them into a form that is suitable for manipulation. The scheduler works on assembly language program codes. Executable binary codes can be disassembled to generate assembly source files.

With assembly source files, the following processing is done:

- Divide the program into segments. Rescheduling of instructions is done within sequential execution blocks only. Thus in this first step, all loops and branch instructions are identified and accordingly segmented. Within these segments, the execution is in sequence. The last flag-modifying instruction before a conditional branch will be fixed to ensure the right flag is prepared for the next conditional branch instruction. Thus after this step, long assembly programs are broken up into relatively short segments.
- Translate into Generic Instructions. The focus of analysis is the inter-dependence between instructions by registers. Thus, a generic instruction (Gins) form will be used for all the instructions. The generic form includes an instruction mnemonic, two source operands and one destination operand. PIns are classified into two groups as shown in Table 2. Interdependence can then be matched by the source and destination operands easily.

Table 2. Generic Instruction Mnemonics.

GIn Mnemonic	Description
Xm	Instructions that do not make use of the multiple FUs. m represents the cycle time.
XXX_AnBm	Instructions that needs n cycles with fast FUs and m cycles with slow FUs XXX represents adder, sub tractor, multiplier or divider.

After these initial processing, the source assembly codes will now be in the form of segments of GIns with two source operands and one destination operand.

2.2.2. *Scheduling Phase*

For each segment from the initialization phase, an algorithm is used to identify or create situations whereby lower-power FUs can be utilized.

Each segment is first scanned to build an inter-dependence table (IT). Each instruction is given a serial number. For a segment of n instructions, an n*n interdependence table is built to indicate the dependence between any two instructions. Independent instructions are then identified and used for resolving stalls.

The algorithm is now applied to the IT as follows: (At the beginning, all instructions are non-fixed and all XXX_AnBm type instructions will be assigned a cycle time of m.)

(i) Find the first non-independent non-fixed instruction A in the segment.

(ii) Non-fixed independent instructions nearby will be moved before or after instruction A so that A will have a foremost position as long as it is valid. The validity of the position must satisfy the following conditions:
   (a) Backward. It is not too close to the previous instruction it is dependent on.
   (b) Forward. It is not beyond the next instruction that is dependent on it.

(iii) If such a position is not found and the instruction that A is dependent on is an XXX_AnBm type that has a faster FU available, use n as its cycle time and repeat (ii).

(iv) Otherwise, all the instruction sequence before this position is fixed and we go back to step (i) to process the next instruction.

The sequence is repeated until all instructions are fixed with a location. In this simple algorithm, independent instructions are like lubricants being moved back and forth so that XXX_AnBm type instructions will have more time to execute before the instruction that is dependent on it arrives and hence slower FUs can be better utilized.

In this algorithm, it is guaranteed that slower FUs are used only when stall does not occur. This is a conservative strategy to ensure the scheduler will never bring down the overall performance. But it is still possible that stalls occur when using fast FUs. In such cases, normal techniques to address hazard are employed, like introducing NOPs or delaying the issuing of the stalled instruction.

With the scheduling algorithm, the actual FU choice for XXX_AmBn type instructions are decided and marked. These instructions will be further mapped to different machine codes for proper FUs in the final object code.

3. Simulation Results & Discussion

To test the efficiency of the scheduling algorithm, a symbolic simulation is designed to determine the probability of applying slower FUs without compromising performance in real programs. The samples tested are x86 assembly codes. Memory addressing modes are taken as register operands for simplicity.

Statistics are collected all the way through the scheduling. Power savings with the proposed ALU is estimated base on the number of instructions assigned to slow FUs and the energy consumption differences between slow and fast FUs (as in Table 1).

Table 3 shows the number of instructions assigned to slow FUs in the target programs and its ratio. The estimated relative energy savings with the proposed ALU is listed alongside.

Table 3 Number of instructions assigned to use slow functional unit and estimated relative energy saving

Program	Addition		Subtraction		Multiplication		Division	
	Number of Slow FU.	Rel. E. Saving.	Number of Slow FU.	Rel. E. Saving.	Number of Slow FU.	Rel. E. Saving.	Number of Slow FU.	Rel. E. Saving.
ARJ	1451 (36.9%)	21.7%	828 (46.3%)	26.8%	78 (29.4%)	12.9%	14 (39.0%)	5.4%
PKZIP	1731 (60.6%)	35.7%	1103 (61.3%)	35.5%	65 (42.8%)	18.8%	12 (60.0%)	8.3%
PKUNZIP	1196 (60.1%)	35.4%	722 (58.5%)	33.9%	53 (51.0%)	22.4%	4 (58.8%)	8.2%
DUNZIP32	302 (17.8%)	10.5%	210 (39.3%)	23.1%	11 (37.9%)	16.7%	3 (22.9%)	3.2%
UNRAR	260 (24.6%)	14.5%	43 (15.2%)	8.8%	2 (12.5%)	5.5%	2 (22.5%)	3.1%
ACE	665 (32.0%)	18.8%	450 (36.0%)	20.8%	32 (18.7%)	8.2%	11 (32.4%)	4.5%

It is clear that a considerable percentage of arithmetic instructions can be run with lower-power slow FUs. This is due to application of the simple yet efficient scheduler which exploits every possible slack in the interdependence between instructions.

More over, it is found that energy savings achieved with adders, subtractors are much more than that with multipliers and dividers. On the other hand, multipliers and dividers take up much larger die areas. Thus, it can be seen that our proposed design is more efficiently applied with simpler operations.

4. Conclusion

In the paper we present a hardware/software co-design for a low-power ALU. The design is based on the fact that fast FUs consume more energy than slower ones with the same function. An assembler level scheduler algorithm is also designed to exploit the slack in the instruction interdependence so as to allow more time for the execution of arithmetic instructions and hence to better utilized the slower FUs.

References

1. Thomas D. Burd and Robert W. Brodersen, "*Design Issues for Dynamic Voltage Scaling*", ISLPED 2000, Rapallo, Italy, Pgs 1 – 6
2. Woonseok Kim, Jihong Kim and Sang Lyul Min, "*A Dynamic Voltage Scaling Algorithm for Dynamic-Priority Hard Real-Time Systems Using Slack Time Analysis*", Proceedings of the 2002 Design, Automation and Test in Europe Conference and Exhibition
3. Chung-Hsing Hsu, Ulrich Kremer and Michael Hsiao, "*Compiler-Directed Dynamic Voltage/Frequency Scheduling for Energy Reduction in Microprocessor*", ISLPED'01, California USA, Aug 6 - 7 2001, Pgs 275 – 278
4. Intel® *Pentium® M Processor on 90nm Process with 2-MB L2 Cache Datasheet*, June 2004.
5. Jessica H. Tseng and Krste Asanovic, "*Banked Multi Ported Register Files for High-frequency Superscalar Microprocessors*", Proceedings of the 30th International Symposium on Computer Architecture, San Diego, California, 2003, Pgs 62-71
6. M. Powell, S. Yang, B. Falsafi, K. Roy, and T. N. Vijaykumar, "Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-Submicron Cache Memories", ISLPED, 2000
7. S. Borka. Design challenges of technology scaling. IEEE Micro, pages 23029, Aug 1999
8. Z. Chen, L. Wei, M. Johnson, K. Roy. "Estimation of Standby Leakage Power in CMOS Circuits Considering Accurate Modeling of Transistor Stacks", ISLPED 1998, pp. 239-244.
9. A. Agarwal, H. Li, K. Roy, "DRG-Cache: A Data Retention Gated-Ground Cache for Low Power", Proc. Design Automation Conf., 2002, pp.473-478.