Optimized register renaming scheme for stack-based x86 operations

Xuehai Qian, He Huang, Zhenzhong Duan, Junchao Zhang, Nan Yuan,

Yongbin Zhou, Hao Zhang, Huimin Cui, Dongrui Fan

Key Laboratory of Computer System and Architecture
Institute of Computing Technology
Chinese Academy of Sciences
{qianxh,huangh, duanzhenzhong,jczhang,
yuannan,ybzhou,zhanghao,cuihm,fandr}@ict.ac.cn

Abstract. The stack-based floating point unit (FPU) in the x86 architecture limits its floating point (FP) performance. The flat register file can improve FP performance but affect x86 compatibility. This paper presents an optimized two-phase floating point register renaming scheme used in implementing an x86-compliant processor. The two-phase renaming scheme eliminates the implicit dependencies between the consecutive FP instructions and redundant operations. As two applications of the method, the techniques used in the second phase of the scheme can eliminate redundant loads and reduce the mis-speculation ratio of the load-store queue. Moreover, the performance of a binary translation system that translates instructions in x86 to MIPS-like ISA can also be boosted by adding the related architectural supports in this optimized scheme to the architecture.

1. Introduction

X86 is the most popular ISA and has become the de facto standard in microprocessor industry. However, the stack-based floating point ISA has long been considered as a weakness of the x86 in compare with the competing RISC ISAs. The addressing of FP stack register is related to a Top-Of-Stack (TOS) pointer. Every x86 FP instruction has implicit effects on the status of the floating point stack, including the TOS pointer and incurs implicit dependencies between consecutive floating point instructions. Furthermore, the stack-based architecture requires one of the operands of an FP instruction comes from the top of the stack, so some transfer or swap operations are needed before real computations.

The AMD x86 64-bit processor attacks the above problems by using a flat register file. It uses SSE2 to replace the stack-based ISA with a choice of either IEEE 32-bit or 64-bit floating point computing precision. However, even the 64-bit mode can not obtain the totally same results as the original double extended FP computing precision. Another fact is that a lot of legacy libraries are written in highly optimized floating point assembly; rewriting of these libraries takes a long time. Therefore, the

replacement of the stack-based ISA is not easy. We should seek for novel techniques to bridge the FP performance gap between x86 and RISC architecture.

This paper first presents a comprehensive and solid methodology of implementing x86-compliant processor based on a generic RISC superscalar core. The techniques to handle x86-specific features such as, complex instruction decoding, Self Modified Code (SMC), non-aligned memory access are outlined. After giving a motivating example, we emphasize on the key elements of our methodology, 2-phase register renaming scheme for attacking the stack-based FP operations. The first phase of the renaming scheme eliminates the implicit dependencies imposed to the consecutive FP instructions by maintaining speculative stack-related information in the instruction decode module. The second phase eliminates almost all of the redundant operations by value "short-circuiting" in rename table, which actually achieves the effect of a flat register file. Critical issues in processor design such as branch misprediction and exception handling are carefully considered.

The scheme has two applications. First it can be used to reduce the mis-speculation ratio of loads in implementing POP instructions in x86 ISA. The results of stores may be directly "short-circuited" to the loads in the renaming module in some cases, so that the redundant loads are eliminated and will not incur mis-speculations in the load-store queue. Secondly, the generic RISC core that originally supports x86 ISA by binary translation can be augmented with the 2-phase renaming scheme, so that the performance of the translated code is boosted. The codes generated by the original binary translation system have a poor performance for FP programs due to the significant semantic gap between the RISC and x86 FP ISA.

The rest of the paper is organized as follows. The design methodology of the prototype processor architecture is presented in section 2. Section 3 provides an example which motivates our optimized scheme. Details about the proposed scheme are presented in section 4. Section 5 briefly summarizes the two applications of the scheme. Section 6 describes the simulation environment and methodology. The simulation results are presented in section 7. The related work is outlined in section 8. We conclude the paper in section 9.

2.Overview of the GodsonX architecture

Fig. 1 shows the architecture of the proposed prototype x86 processor GodsonX which is built based on the core of Godson-2C[1] core, a typical RISC 4-issue out-of-order superscalar. The architectural supports for x86 architecture are presented as shaded blocks.

The front end of x86 processor is much more complex than the RISC processor. It is organized in 5 pipeline stages. The first stage generates instruction addresses. The second stage is composed of five modules providing the content of instructions, pre-decode information and performing branch prediction. The third stage, decode-0, consists of four sub-modules and is responsible for instruction alignment and queue management. The decode-1 module reads the aligned x86 instructions from the first instruction register (IR1 in the figure), translates them into the "RISC style" intermediate instructions and some micro-ops, and puts them in the second instruction

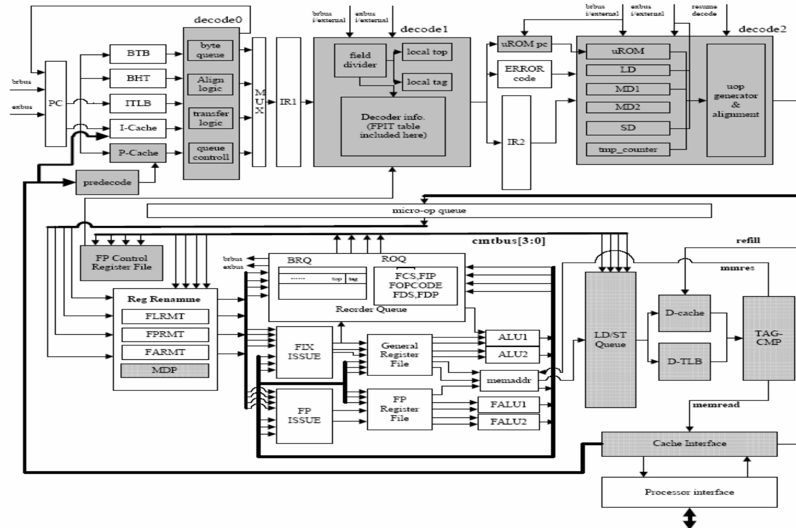

Fig. 1. GodsonX architecture

register (IR2 in the figure). The decode-2 module generates sequences of micro-ops and put the sequences in the micro-op queue (uop in the figure). The methods that decode-2 uses to generate micro-op sequences include: short micro-op sequence Decoder (SD), Long micro-op sequence Decoder (LD), Memory access Mode decoder (MD) and micro-op Rom (urom). The uop queue acts as a buffer between decode-2 and the register rename module inside the RISC core, enabling stable instruction issue from the front end.

The architectural supports to the FP stack are distributed. These modules work cooperatively to guarantee the correct stack status while ensuring efficient execution of floating point instruction. The decode-1 module maintains a local copy of TOS and Tag, and has a table storing the x86 FP instruction information. Each micro-op should carry the TOS after dispatched from the register renaming stage. The reorder queue (ROQ) handles FP exceptions specified in x86 ISA. The branch queue (BRQ) keeps the local TOS and Tag of the branch instruction for misprediction handling. The floating point data type (Tag) of the result value is computed in FALU1/2, and passed to ROQ in write back stage. When a floating point instruction commits, ROQ writes the TOS and Tag into the architectural FP status and tag word. Under exceptions, the decode-1 module recovers the TOS and Tag from these registers.

We tried to handle some burdensome x86 ISA features, such as Self Modified Code (SMC) and non-aligned memory accesses, etc. by architectural modifications or supports. The method to handle SMC is in the granularity of cache line, so it has the advantage over the method adopted by Intel, which detects and handles the event in the granularity of page. Specifically, we added a Simplified Victim Cache (SVIC) to the decode module. As a store writes back, we use the address to look up the ICache and SVIC, if an entry is found, then an SMC occurs, then the pipeline need to be flushed. Similarly, the miss queue is also looked up as a store writes back. When the ICache misses, it is also needed to look up the data in the DCache, while sending the

request to miss queue, trying to find the data in memory. Similarly, it is also necessary to look up the data in the store queue when ICache misses, since store queue serves as a buffer between pipeline and DCache. In our processor, the L2 and L1 cache are exclusive. If a missed L1 access hits a dirty line in L2 cache, L2 cache should also write the line back to memory, otherwise the data will be lost. As an optimization, we modified the former memory access architecture in our processor and proposed an efficient way to execute non-aligned memory accesses in x86. In the new architecture, the LSQ(LD/ST Queue) is placed in the position before the DCache and DTLB accesses. In this way, the latency in cache tag compare stage is reduced, since the load and store dependencies are only needed to be checked when the loads/stores are issued from the LSQ. We also leverage this architecture to execute non-aligned memory access efficiently, this kind of operations are split into two operations after it goes through the LSQ, and the first access can be interleaved with the address calculation of the second one. The 80-bit memory accesses in FP instructions can be handled in a similar way. Moreover, we found that segment register is usually not changed as x86 program executes, so we can speculate on its value, which makes the address calculation simpler. The performance of memory access module can also be optimized by making the common operations run faster, which reduces the hardware cost. We found that in 2 issue memory access pipeline, one-port TLB is enough in the common case. The MDP in register renaming module is a simple Memory Distance Predictor, which can reduce the possibility that the pipeline flush due to load/store conflicts.

3. Motivating Example

A floating point computation example is presented in this section, showing the advantages of the optimized scheme. Consider the computation: atan((a+b)/(a*c)), a possible x86 instruction sequence and two possible corresponding micro-op sequences are given below.

X86 instructions		naïve micro-op sequence	optimized sequence
FADD	ST(1);	fadd fr(5), fr(5), fr(6)	fadd fr(5), fr(5), fr(6)
FXCH	ST(4);	fmov fr(9), fr(5)	fxch fr(5), fr(1)
FMULP	ST(1),ST(0);	fmov fr(5), fr(1)	fmul fr(6), fr(6), fr(5)
FLD	ST(3);	fmov fr(1), fr(9)	fmov fr(5), fr(1)
FPATAN;		fmul fr(6), fr(6),fr(5)	fpatan fr(5), fr(5), fr(6)
		fmov fr(5), fr(1)	
		fpatan fr(5), fr(5), fr(6)	

Fig. 2 shows the change of FP stack status when the instructions are executed. We assume the left-most the initial state, where TOS is 5. The FXCH instruction swaps the content of ST(4) and ST(0). The suffix "P" in the multiplication instruction implies that a "pop" operation is required. The "pop" increases the TOS and empties the former top element. The last instruction computes the arc-tangent function of (ST(0)/ST(1)), the source operands position are fixed, so there is no need to specify them in instruction. It reflects a feature of stack-based architecture. Also, the FPATAN instruction includes a "pop" operation.

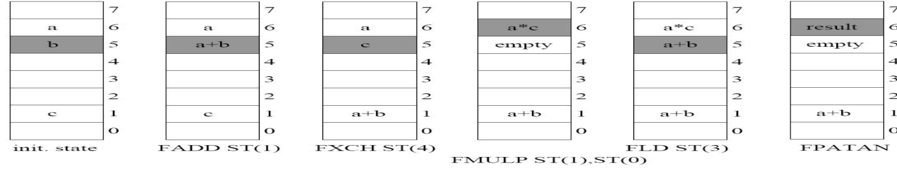

Fig. 2. Stack operations of the x86 instructions

We use four FP register number address spaces. FP registers used in FP instructions are called **relative FP stack registers** (ST(i), $i \in [0,7]$), the relative FP registers are mapped to **absolute FP stack registers** (st(i), $i \in [0,7]$) by adding ST(i) to TOS (module 8). The FP registers used in micro-op are **FP logical registers** (fr(i), $i \in [0,15]$). The FP registers after renaming in the RISC core are **FP physical registers** (pr(i), $i \in [0,63]$). st(0)-st(7) are directly mapped to fr(0)-fr(7). Temporary registers are used to hold intermediate results. The micro-op sequences can use up to 8 temporary FP registers which are fr(8)-fr(15). In the naïve sequence, we use three transfer operations ("fmov") to implement FXCH. In the optimized scheme, a dedicated swap operation ("fxch") is used.

Table 1 and table 2 show the execution and register mapping processes of the two micro-op sequences. We find that the number of micro-ops that would go into the issue queue is 7 in the naïve sequence and 3 in the optimized sequence; while the numbers of physical registers consumed are 10 and 6, respectively. Both the micro-ops and the mapped physical registers are reduced. Due to the nature of stack-based architecture, there are inherently a large amount of FXCH instructions in x86 FP program. Reducing the number of this kind of operations can directly reduce the execution time of the program. Moreover, it will alleviate the burden to the physical register file and issue queue, which makes other operations execute faster as well.

In the next section, we will present our novel 2-phase renaming scheme. The first phase of renaming releases the serial requirement in decoding x86 FP instructions via speculative local copy of certain floating point information in decode-1 module. The second phase adopts an optimized RAM-based approach, which can support the optimization.

Table 1. Register mapping and execution of the naive micro-op sequence

Map-Table	orig. inst	renamed inst	mapping	executed ops
fr1->pr1,fr5->pr5, fr6->pr6	fadd fr5,fr5,fr6	fmul pr9,pr5,pr6	fr5->pr9	(pr5+pr6)->pr9
fr1->pr1,fr5->pr9, fr6->pr6	fmov fr9,fr5, fmov fr5,fr1 fmov fr1,fr9	fmov pr10,pr9 fmov pr11,pr1 fmov pr12, pr10	fr9->pr10 fr5->pr11 fr1->pr12	pr9->pr10 pr1->pr11 pr10->pr12
fr1->pr12, fr5->pr11, fr9->pr10, fr6->pr6	fmul fr6,fr6,fr5	fmul pr13, pr6, pr11	fr6->pr13	(pr11*pr6)->pr13
fr1->pr12, fr5->pr11, fr9->pr10, fr6->pr13	fmov fr5,fr1	fmov pr14, pr12	fr5->pr14	pr12->pr14
fr1->pr12, fr5->pr14, fr9->pr10, fr6->pr13	fpatan fr5, fr5, fr6	fpatan pr15, pr14, pr13	fr5->pr15	atan(pr13/pr14) ->pr15

Table 2. Register mapping and execution of the optimized micro-op sequence

Map-Table	orig. inst	renamed inst	mapping	executed ops
fr1->pr1,fr5->pr5, fr6->pr6	fadd fr5,fr5,fr6	fmul pr9,pr5,pr6	fr5->pr9	(pr5+pr6)->pr9
fr1->pr1,fr5->pr9, fr6->pr6	fxch fr5, fr1	**eliminated**	**fr5->pr1** **fr1->pr9**	Swap the FP physical register that fr5 and fr1 mapped
fr1->pr9, fr5->pr1, fr6->pr6	fmul fr6,fr6,fr5	fmul pr10, pr6, pr1	fr6->pr10	(pr1*pr6)->pr10
fr1->pr9, fr5->pr1, fr6->pr10	fmov fr5,fr1	**eliminated**	**fr5->pr9**	Make fr5 map to the physical register that fr1 mapped to, it is pr9
fr1->pr9, fr5->pr9, fr6->pr10	fpatan fr5, fr5, fr6	fpatan pr11, pr9, pr10	fr5->pr11	atan(pr10/pr9) ->pr11

4. Optimized 2-phase register renaming scheme

4.1 Mapping from stack registers to logical registers

In the first phase mapping, the FP stack registers are mapped to FP logical registers. We adopt a speculative decoding technique in this process. The decode-1 module maintains a local copy of partial TAG and TOS. The TAG is partial since it just indicates if a FP register is empty. The decode stage determines the absolute FP register based on the local TOS and update these information after each FP instruction is decoded, according to the specifications of each FP instruction. In this way, the decoding of FP instructions can be pipelined, since the decode module does not have to wait for the committed TOP and TAG information. Here we should note that the effects of each FP instruction on the stack are totally predictable.

If no exceptions, the local TOS and Tag is synchronized with the architectural TOS and Tag in the FP status and tag word when the instruction commits. In case of exceptions, the changes to the local TOS and Tag at the decode stage have to be recoverd to the architectural state. We explain the branch misprediction case by an example.. Fig. 3(a) presents a branch misprediction scenario. The nodes 1 and 2 represent the committed instructions, the nodes 3 and 4 represent the instructions executed in the correct path but not committed, node 4 is the branch instruction, and nodes 5-7 are the instructions in the wrong path, executed and need to be cancelled. From this scenario, if the TOS and Tag are recovered from the FP status word and the Tag word, the decode stage will hold TOS and Tag information of the last instruction committed before the branch, that is node 2. But this is incorrect, since what we need is the TOS and Tag after the execution of the branch instruction, the node 4. Therefore we need to keep the TOS and Tag of each branch instruction in the BRQ shown in Fig. 1. When a branch misprediction occurs, the decode stage recover the TOS and Tag from BRQ.

The partial TAG information can be used in the detection of stack overflow/underflow. An FP stack underflow occurs when an instruction references an

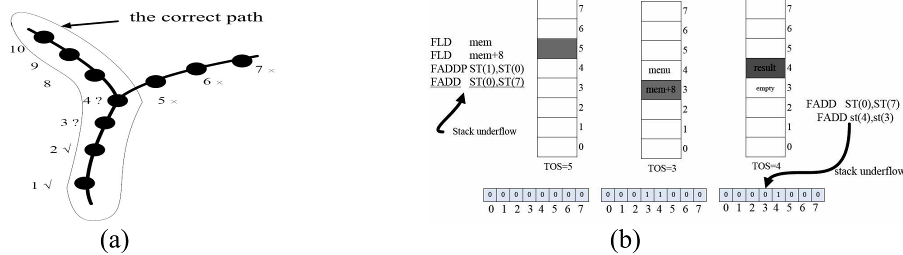

Fig. 3. (a) branch misprediction scenario. (b) An example: detection of FP "stack underflow"

empty FP stack register, a stack overflow occurs when an instruction loads data into a non-empty FP stack register. Fig. 3(b) shows an example of detecting FP stack underflow conditions. The partial TAG is the 8-bit vector at the bottom; "0" indicates the related position in FP stack is empty, "1" means the position is non-empty.

The Floating Point Instruction Table (FPIT) is used as an effective and low cost way to maintain the information of each FP instruction in decode module. After analyzing the bit codes of each instruction, we found certain group of instructions has similar bit codes and similar effects to the stack. We can represent information for these FP instructions by just one entry in FPIT. This technique makes the table smaller. In each entry, we store the effects to the stack. More details about this table are out of the scope of the paper.

4.2 Optimized register mapping in the RISC core

There are two ways to implement the register renaming, the RAM-based design and CAM-based design, they use separate or merged architectural and rename register files. The former design of Godson-2C processor adopts the CAM approach, which can not support the optimization in the paper. We propose a new RAM-based register renaming design. This design allows one physical register to be mapped to more than one logical registers. Fig. 4 gives the outline of this architecture.

We use three mapping tables to maintain the relationship between FP logical registers and physical registers. The Floating point Logical Register Mapping Table (**FLRMT**) is used to rename logical registers to physical registers. It has 16 entries representing 16 floating point logical registers. The field "pname" indicates which physical register the logical register is mapped to. Each FLRMT entry contains 8 lastvalid items, corresponding to 8 BRQ entries. The lastvalid(i) keeps the mapped physical register number when the branch instruction in BRQ(i) is mapped. The Floating point Physical Register Mapping Table (**FPRMT**) merely maintains the state of each physical register. It has 64 entries corresponding to 64 physical registers. There three fields in each entry. The state records the state of the physical register, brqid is used in branch misprediction to recover the correct register mapping, and counter indicates how many logical registers are mapped to this physical register. When the register renaming stage finds an entering instruction an "fmov", it directly maps the destination register to the physical register of the source register, and increases the counter for the physical register. When "fxch" is encountered, the source

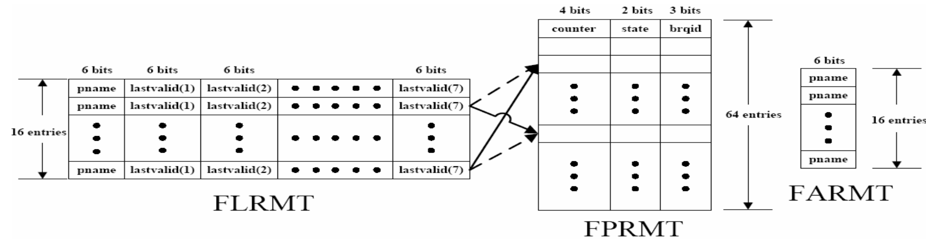

Fig. 4. optimized register renaming inside RISC core

and destination registers are simply swapped. When an instruction committed, the counter for the destination physical register is decreased. We need four bits for the counter field, since at most 16 logical registers can be mapped into a physical register. Note that the change of mapping between logical and physical registers at the register renaming stage by fmov or FXCH is speculative; the instructions can be canceled later because of exception. The Floating point Architectural Mapping Table (**FARMT**) records the committed physical register which the logical register is mapped to, it is used in exception recovery. This table has 16 entries corresponding to 16 logical registers. Each entry just records the physical register that the logical register mapped to. The table is updated when an instruction is committed. When an exception is encountered, the renaming stage can recover the mapping relationship from FARMT. This optimized architecture simplifies lookup logic compared with the CAM-based implementation and is more scalable.

TAG update should also be considered in the optimization. When an instruction committed, the tag for the destination register should be updated, reflecting the latest status. In our design, the tag is computed in FALU, but the eliminated "fxch" and "fmov" will not enter FALU. This is not a problem for "fxch", since its two operands are both architectural visible registers, so the only thing to do as an instruction commits is to swap the tags for the two operands in the FP tag word. For "fmov", it is more difficult because the source operand may be a temporary register. To make the design simple, the optimization is not applied to the special case. As the statistics data shows, the special case is rare.

5. Applications of the renaming scheme

The proposed scheme has two applications. First, it can be used as the supports for the Godson-2C processor that implements MIPS-like ISA and runs application-level binary translator to support x86 applications. The binary translator that we conducted the experiments is Digital Bridge[6]. It works in a Godson based LINUX server, and translates the elf file of x86 ISA to Godson ISA (MIPS-like). Although the translator works well for fix point programs the performance of floating point applications suffers. It is mainly due to the remarkable ISA semantic difference between x86 and general propose RISC in floating point specification. The existing method on binary translation is not efficient enough to bridge such a gap, architectural supports are needed to narrow the gap. Without architectural supports, the Bridge translator use

static FP registers in Godson processor to emulate FP stack operations. When loading data into the FP stack register, for example, ST(2), we must dynamically determine the corresponding absolute register and put the value in fr(2). The process will incur a lot of swap operations in the target Godson code. This approach still needs the help of memory. The valid values on the stack should be loaded from and stored into memory at the beginning and the end of each basic block. Finally, this approach assumes that the TOS is the same and TAGs are all valid at entry of each basic block. Only under this assumption, ST(i) can always correspond to fr(i), regardless of the preceding path from which the code arrives the entry. But a large amount of extra code must be added at the head of the translated code for each block to judge if the above speculation is held. From the experiment results, the condition is satisfied almost all the time. It is obviously a waste to execute a large segment of extra code for rare conditions. We add our architectural support for FP stack to Godson-2C processor without x86 features. With these supports, we can directly use the relative FP registers in the translated code, making the burden of maintaining status of FP stack to hardware.

As the second application of the method, it can also used to eliminate redundant loads. An impediment to Godson-X performance is the high miss rate of load speculation. After analyzing the program execution behavior, we found that the problem came from the x86 PUSH and POP instructions for parameter passing in function calls. These two instructions are mapped to store and load micro-ops. In a function call, the store and load come in pair and close to each other. Godson-X always speculates on the value of the load before the store commits. Therefore when the store commits, mis-speculation occurs. We added a 4-entry table to the register renaming module to forward the store value to the loads. The table maintains the source register numbers and memory addressing information of the 4 most recent store instructions. If a load instruction's memory address matches to one of the entries, it can be eliminated by modifying the register mapping relationship to directly get the stored value. Moreover, we are trying to extend this technique to eliminate redundancy in control flow with some hardware support. It is out of the scope of this paper.

6. Experimental Infrastructure

We have developed a cycle accurate full-system simulator for x86-compliants. Unlike the Simplescalar-based performance simulators, which decouple the execution and timing logic and can only provide an estimation of the performance, our simulator models the exact signals and timing except inside the ALU/FALU. This makes the result more accurate. Table 3 shows the detailed configuration of the simulator. For the latency of FP operations, we make following assumptions on latency: absolute, negation, comparison and branch take two cycles; addition, subtraction and conversion take three cycles; multiplication takes four cycles; division and square root take 4 to 16 cycles to complete; and transcendental functions take 60 cycles to complete. The real computation is carried out by a modified library of standard FP software implementation, the main modifications are FP exception handling.

Table 3. Configuration of GodsonX processor

decode width	at most 2 x86 instructions each cycle
functional units	2 fix point ALU, 2 floating point ALU, 1 memory
ROQ	32 entries
BRQ	8 entries
fix issue queue	16 entries
float issue queue	16 entries
branch predictor	Gshare: 9-bit ghr, 4096-entry pht, 128-entry BTB,direct mapped
L1-ICACHE	64KB 4-way set associative
L1-DCACHE	64KB 4-way set associative
memory access latency	50 cycles for the first sub block, 2 cycles for consecutive sub blocks

We use X86 emulator Bochs[9], which can boot LINUX and Window XP, as a reference in validating our design. Every time an instruction is committed, the whole architectural state is compared with Bochs. Due to this method, we debugged and validated our design. Finally, our simulator can boot the LINUX and Window XP.

SPEC CPU2000 is used as our benchmark. First, we find the representative region of each program by a SimPoint-like performance simulator for GodsonX processor, which is built from the counterpart for Godson-2C processor[5]. We fast-forward each program to its representative region and run 1 billion cycles using the cycle-accurate simulator to get precise results.

7. Simulation results and Discussion

7.1 Performance and characteristics of x86 programs

Fig. 5(a) presents the performance comparison between our processor and the 2.4 GHz Intel Celeron processor. The IPC of the latter is obtained as follows. We first run each program in Bochs and record the instruction count, them we execute it in a real Celeron machine, record the execution time. We compute the IPC of each program and then compare it with the IPC of representative region in GodsonX. For some programs such as wupwise, swim, facerec and swim, performance of GodsonX is much better than that of Celeron, for programs like applu, equake, ammp and apsi, performances are similar. However, for some programs, especially sixtrack, GodsonX's performance is worse. Fig. 5(b) shows the latency distribution of micro-ops in GodsonX, the height of each bar represents the absolute number of cycles from map to commit. We can see that most of the cycles are due to register mapping or waiting for commit. This indicates that a large physical register file or reorder buffer is needed. Fig. 5(c) shows the cycle distribution with respect to the number of micro-ops committed in a cycle. For every program more than one micro-op is committed each cycle on average, especially for ammp, in most of the time four micro-ops are committed each cycle. It indicates that the efficiency of GodsonX is quite good. Fig. 5(d) presents the average number of micro-ops per x86 instruction, which indicates the quality of our micro-op mapping. On average about 2 are needed to implement an x86 instruction.

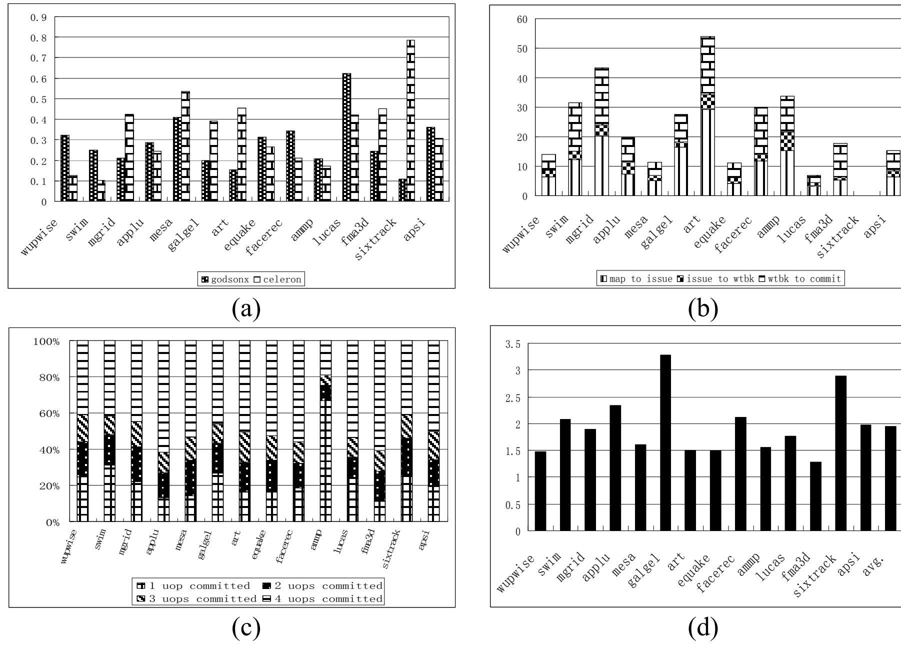

Fig. 5. Execution results of x86 program on our processor

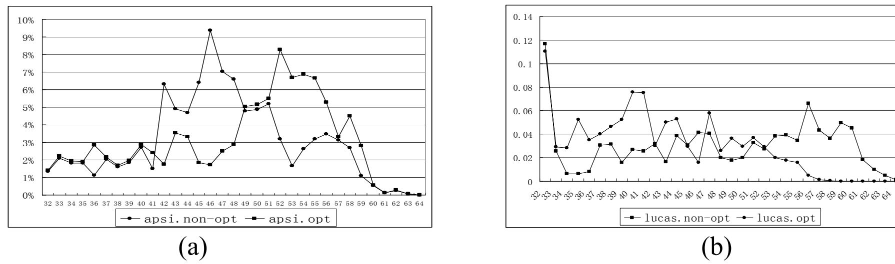

Fig. 6. Comparisons of register renaming

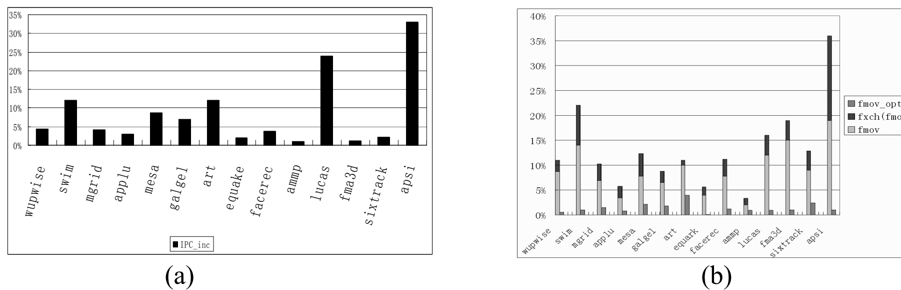

Fig. 7. IPC increase and the ratio of eliminated operations

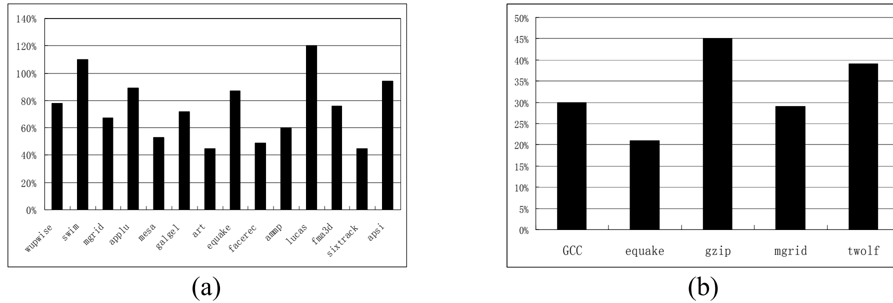

Fig. 8. Effects on the binary translation system and the ratio of eliminated mis-speculation

7.2 Effects of the optimized scheme

Fig. 6 shows the effect of the optimization in mitigating the burden to register renaming. The information is indicated by the percentage of total execution time during which certain number of renaming table entries are occupied. We find that after optimization, the curves shift left. This means that fewer entries are used in certain percentage of time. Fig. 7(a) shows the improvement to IPC. We can see that both apsi and lucas get a big IPC increase, as high as 30%. This observation is consistent with the optimization effect on usage of the renaming table shown in Fig.6. Fig. 7(b) shows the number of “fmov” before and after the optimization. The elimination of “fmov” comes from two sources. First, the “fmov” for FXCH are completely eliminated by simply swapping the source and destination registers mappings. Secondly, a large portion of other “fmov” resulted from FLD or FST are eliminated by the register renaming module. We can see that more than 10% of micro-ops are removed on average. In Figure 8, we show the impacts of the application of technique. From (a) we can see the performance of the binary translation system boosts significantly; this is due to the augmented architecture on which the code generated by the binary translation executes. In this architecture, certain features of the floating point stack are incorporated, so that the binary translation system can easily generate simple and efficient code for the x86 FP application. In (b), the ratio of the eliminated load mis-speculations are presented, the mis-speculation is frequently in SPEC2000 Integer programs, so we show the ratios for several integer programs.

7.3 Hardware costs comparison

This section we show two alternative implementations of the register renaming module. One is the design proposed in this paper, the other is the register renaming module in Godson-2C. The results in Table 4 are derived from Synopsys Design Compiler with 0.13um standard cell library for TSMC. We can see from the comparison that the critical path of the optimized scheme is slightly longer than the

Table 4. Hardware cost comparison

	Lat. (ns)	Area(um^2)
GodsonX	1.25	616841.937500
Godson-2C	1.23	981161.687500

design of Godson-2C, but the area comsumed by the GodsonX register renaming module is greatly reduced. The main reason to the decrease is in the Godson-2C design, a large combinational logic is used to generate a table that maps the logical registers to physical registers, this part of logic consumes a lot of area. Although the proposed scheme has a longer critical path than that of Godson-2C, the RAM-based design apporach has better scalability. When the number of physical registers increases, the proposed scheme will show more advantages over the former design.

8. Related work

The implementation of FP stack is a critical issue in x86-compliant processor design. Some mechanisms have been patented by Intel[4] and AMD[3], but they are different from the scheme proposed in this paper. The main distinctions are that they normally adopt multiple tables to hold the stack related information, and the structures to hold the information are distributed in the processor. The synchronizations under exceptions and branch mis-predictions are much more complicated. The modification to the RISC core in our scheme is trivial and the handling of exceptions or mis-predictions is easy to understand and implement. More important is that we present an applicable methodology of implementing the FP stack based on a generic RISC core efficiently.

The elimination of FXCH has been used in some x86 processor, but it can only be done when FXCH comes with certain types of instructions, in those conditions, it can be combined with the surrounding instructions and eliminated. Both Intel and AMD employ a dedicated unit to execute FXCH instruction. Our scheme is more general and has lower cost. We only incorporate some simple functions in register renaming stage to detect the optimization opportunities. As in our scheme, not all FXCH in Intel or AMD processors can be eliminated. For example, under stack error (stack overflow or underflow), AMD processor will generate 5 micro-ops for the FXCH instruction. Moreover, from P4 processors this operation will have 3-cycle execution time again. Due to the elegant style of eliminating such operations in our scheme, the optimization will exist continually in our processor.

Our attempt is the first effort to implement a full x86-compliant processor based on a typical RISC core. The methodology presented in this paper can be applied to build processor in different ISAs. We also provide some x86 program characteristics and behaviors on our processor. IA-32 execution layer[7] and transmeta morphing software[8] are two efforts to translate x86 programs to other ISAs. Software based approaches are adopted in these systems, and the underlying architectures are VLIW, while our methodology is based on hardware architectural support to an existing and more general superscalar architecture. It is also the first work to investigate the impact of architectural support to binary translation.

9. Conclusion

This paper presents an optimized floating point register renaming scheme for stack based operations used in building an x86-compliant prototype processor. We compared the hardware cost of two register renaming designs; the proposed scheme has a slightly longer critical path but greatly reduced area. We find a large amount of swap and data transfer instructions in FP programs, and most of them can be eliminated by our proposed scheme. The IPC improvements due to the optimization are as high as 30% for some programs, and near 10% on average. Similar techniques in the scheme can also be extended to eliminate redundant loads and used as the architectural supports for RISC superscalar core to boost the performance of the binary translation system which run on that architecture. Our future work includes finding the optimal design trade-off in the co-designed x86 virtual. We will implement the x86 features that are critical to the performance and easy to be supported in hardware, for example the supports for the floating point stack, while implementing the complicated and unusual features in software.

Acknowledgement

This work is under the support of the National Basic Research Program, also called 973 program in China (grant number: 2005CB321600) and Innovative Program of ICT (grant number: 20056610).

References

1. Weiwu Hu, Fuxin Zhang, Zusong Li. Microarchitecture of the Godson-2 processor. Journal of Computer Science and Technology, 3 (2005) 243-249.
2. David Patterson, John Hennessy. Computer Architecture: A Quantitative Approach. Morgan Kaufmann Publishers, Inc. 1996.
3. Michael D. Goddard, Scott A. White. Floating point stack and exchange instruction. US Patent Number: 5,857,089. Jan.5, 1999
4. David W. Clift, James M. Arnold, Robert. P. Colwell, Andrew F. Glew. Floating point register alias table fxch and retirement floating point register array. US Patent Number: 5,499,352. Mar.12, 1996
5. Fuxin Zhang. Performance analysis and optimization of microprocessors. PHD Thesis, Institute of Computing Technology, Chinese Academy of Sciences. (6) 2005
6. Feng Tang. Research on dynamic binary translation and optimization. PHD Thesis, Institute of Computing Technology, Chinese Academy of Sciences. (6) 2006
7. Leonid Baraz, Tevi Devor, Orna Etzion, Shalom Goldenberg, Alex Skaletsky, Yun Wang, Yigai Zemach. IA-32 execution layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium®-based systems. MICRO-2003 (11) 2003
8. Dehnert. J.C., Grant. B.K., Banning. J.P, Johnson, R.; Kistler. T., Klaiber. A., Mattson. J., The transmeta code morphing software: using speculation, recovery, and adaptive retranslation toaddress real-life challenges. CGO-2003 (3) 2003
9. Bochs: The Open Source IA-32 Emulation Project . http://bochs.sourceforge.net/