Doubling the Number of Registers on ARM Processors

Hsu-Hung Chiang, Huang-Jia Cheng, and Yuan-Shin Hwang
Department of Computer Science and Information Engineering
National Taiwan University of Science and Technology
Taipei 106
Taiwan

***Abstract*—It is critical that more architectural registers are available to the compiler and programmer, as a small number of architectural registers might hinder the compiler and programmer from producing efficient code. Although modern chip manufacturing processes could easily put many registers on an ARM processor, only 16 general registers are accessible by the compiler and programmer. Doubling the number of architectural registers requires adding another bit to each register field of instructions, and hence increases code size. One approach has been developed to use the underutilized condition bits to double the number of architectural registers from 16 to 32 but maintain the width of ARM instructions. Although performance gain can generally be achieved by most of benchmark programs on a 32-register ARM, this approach would still hurt performance for programs with high ratios of instructions that can be conditionalized. Therefore, a better approach would to keep the conditional execution feature even when the 4-bit condition field is used to encode the extra registers. This paper proposes to borrow the IT instruction from Thumb II ISA to represent the predicates of conditionalized instructions on the 32-register ARM. GCC and SimpleScalar/ARM have been modified to handle the new instruction format and the revised IT instruction, and experimental results have shown that performance can be improved by 10.4% on average for MediaBench II benchmarks on the 32-register ARM with conditional execution.**

***Keywords*-Conditional Execution, ISA, Instruction Encoding, Registers**

I. Introduction

The number of architectural registers exposed to the compiler and the programmer is crucial to the code quality, as a small number of architectural registers might restrict compilers from effectively performing compilation and optimization or prevent programmers from storing some frequently used data [16]. Therefore, it is critical that more architectural registers are available to the compiler and programmer without expanding the code size significantly. Although modern chip manufacturing processes could easily put hundreds of registers on a chip, only a small number of them are accessible by the compiler and programmer since the number of architectural registers is determined by the width of the register field in an instruction. For instance, each register field in ARM instructions has 4 bits and hence it limits the compiler and programmer to access only 16 architectural registers although ARM has more physical registers [14].

Doubling the number of architectural registers requires adding another bit to each register field of instructions. Since an ARM instruction has at most four register operands, e.g. `ADD R1, R3, R5, LSL R2`, it would expand by up to four bits when the number of registers are doubled from 16 to 32. If every instruction is stretched from 32 bits to 36 bits in order to accommodate these extra bits, the code size would increase by about 12% on the 32-register ARM. In addition, widened instructions would intensify the I-cache and memory traffic, and hence bump up the power consumption [15], [16].

A better approach is to double the number of architectural registers but maintain the width of ARM instructions. Specifically, up to four unused bits need to be allocated from each ARM instruction. However, it is generally impossible to identify enough redundant bits to accommodate these extra bits for the expanded register fields. Therefore, identifying underutilized bits would be a feasible solution. One possible candidate is the 4-bit condition field in every ARM instruction that specifies the predicate of conditional execution, as ratios of conditionalized instructions are generally very low [4].

Conditional execution (or predicated execution) is an instruction set architecture (ISA) feature, which predicates individual instructions on processor status flags [5], [8], [13]. Every ARM instruction can be made to execute conditionally [14]. An instruction has only its normal effect if the status satisfies a condition specified in the instruction, and acts as a NOP if otherwise. As ARM has deployed a straightforward orthogonal instruction coding that all instructions can be coded conditionally on all situations, every instruction reserves a 4-bit field for the predicate. Besides, the instruction space for conditions is underutilized since only small percentages of instructions are actually conditionalized in modern embedded applications like interactive multimedia communications, which generally demand massive computations [2]. Therefore, the condition field in every ARM instruction is a good candidate to house the extra bits for the expanded registers. In other words, this approach looks to trading conditional execution for more registers on ARM.

Trading conditional execution for more registers is a good compromise since performance gain can generally be

achieved by most of benchmark programs on a 32-register ARM [4]. However, performance loss can still be observed for those programs with high ratios of instructions that can be conditionalized, that is, conditional execution is still a vital feature for certain applications. For instance, MediaBench benchmark suite [12] encounters a minor slowdown of 2.3% on a 32-register ARM with no conditional execution. Therefore, it would be beneficial to keep the conditional execution feature even when the 4-bit condition field in every ARM instruction is used as the extended register fields to encode the extra registers. This paper proposes to borrow the IT instruction from Thumb II ISA to represent the predicates of conditionalized instructions on the 32-register ARM.

GCC [7] and SimpleScalar/ARM [3] have been modified to handle the new instruction format and the revised IT instruction. Performance improvement can generally be expected since register spills can be significantly reduced by storing more values in the extended register file, while maintaining the ability of conditional execution. Experimental results have shown that the perform loss seen on the 32-register ARM for MediaBench can be turned to performance gain when conditional execution is reactivated. In addition, performance improvement for MediaBench II benchmarks is 10.4% on average on the 32-register ARM with conditional execution.

II. Trading Conditional Execution for More Registers

All ARM instructions can be made to execute conditionally [14]. Specifically, an instruction only has its normal effect if a condition specified in the instruction matches the N (Negative), Z (Zero), C (Carry) and V (Overflow) flags in the *Current Program Status Register* (CPSR), as shown in Figure 1. If the flags do not satisfy this condition, the instruction acts as a NOP. This conditional execution feature allows small sections of if- and while-statements to be encoded on ARM without the use of branch instructions, increasing the effectiveness of pipelined execution by avoiding pipeline stalls caused by branch operations.

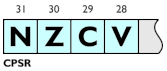
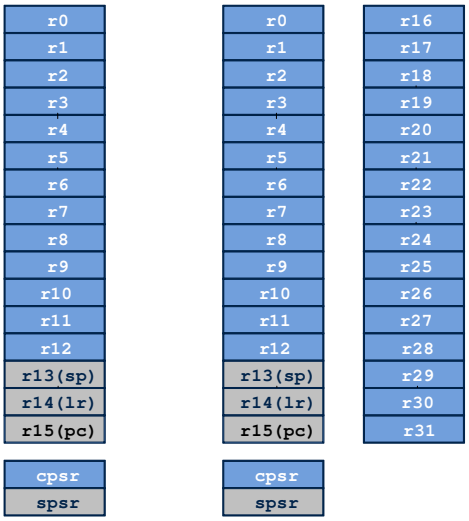


Figure 1. Status Flags

A. Instruction Encoding

The conditional execution feature is implemented with a 4-bit condition code field (i.e. bits[31:28]) on every instruction, as shown in Figure 2. This cuts down significantly on the encoding bits available for displacements in memory access instructions or registers in data processing instructions. An instruction encoding scheme has been developed to use this 4-bit condition field to represent registers higher than r15 [4]. The original register file containing 16 general-purpose register shown in Figure 3(a) will be expanded to

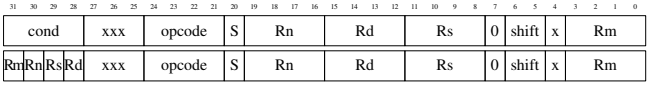
Figure 2. Condition Encoding



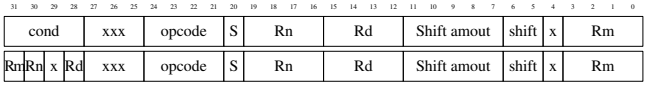
(a) Original (b) Extended Register File

Figure 3. Registers

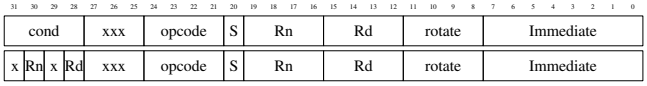
32 registers by the new instruction encoding, as depicted in Figure 3(b). The first 16 registers of the extended register file remain exactly the same as the original registers.



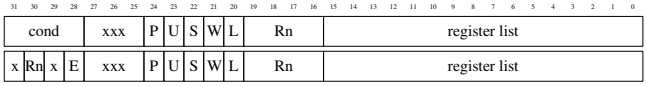
(a) 4 Register Fields



(b) 3 Register Fields



(c) 2 Register Fields



(d) 1 Register Field

Figure 4. Instruction Formats

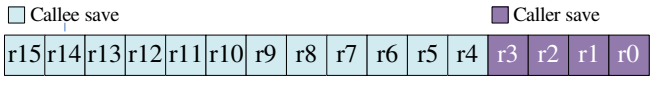
As there are at most four register fields in an ARM instruction, such as the data processing register shift instruction ADD R5, R5, R3, LSL R2, each register field can take one bit from the condition field and that is why the number of ISA registers can only be doubled from 16 to 32. Figure 4(a) shows the instruction formats with four register

fields. The top format is the original instruction encoding format that represents an ARM instruction with the form of OP Rd, Rn, ,Rm, SHIFT Rs. The bottom format in Figure 4(a) depicts that each bit of the 4-bit condition field is assigned to one of the four register fields in the instruction. As a result, each register field in the new instruction format has 5 bits and hence can address all the 32 registers in the extended register file.

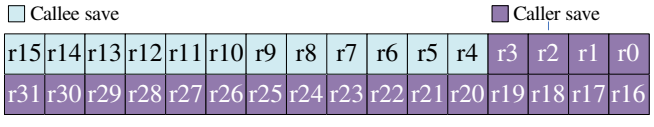
Figure 4(b) and Figure 4(c) illustrate that the instruction formats with three and two register fields can be easily modified to accommodate the extra bits for the extended registers. However, special attention is needed when changing the last instruction format with only one register field displayed in Figure 4(d). The reason is that this instruction format is used to encode the load and store multiple instructions, which can load and store a subset, or possibly all, of the general-purpose registers from and to memory. Unfortunately, it is impossible to represent all the 32 registers in the extended register file in the 32-bit instruction format. Therefore, the extended register file will be divided into two halves: the lower half (r0-r15) and the upper half (r16-r31), and only half of the registers can be accessed by a load/store multiple instruction. Specifically, when the "E" bit is cleared, a load/store multiple instruction can only reference the lower half of registers. When the "E" bit is set, a load/store multiple instruction can only access the upper half of registers.

B. GCC Retargeting

GCC 4.4.1 has been retargeted to generate ARM binary code with the new instruction encoding, which supports 32 ISA registers. GCC must be informed that the number of registers has been expanded to 32, which requires several parameters in the files *arm.h* and *aout.h* to be updated. In addition, the sets of caller save registers and callee save registers have to be modified as well. Figure 5(a) shows that r4-r15 are callee save registers while r0-r3 are caller save registers in the original GCC setting. In order to keep things simple and avoid pushing too many register values onto the stack by procedure invocation, the set of callee save registers remains the same and all the extended registers r16-r31 are assigned as caller save registers.

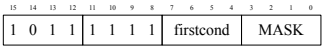


(a) 16 Registers



(b) 32 Registers

Figure 5. Caller and Callee Save Registers



(a) IT Instruction

(b) 32-Bit IT Instruction

Figure 6. IT Instructions Formats

III. Reactivating Conditional Execution

Similar to the 32-register ISA described in the previous section, the Thumb 16-bit encoding space does not have sufficient space to retain this ability and therefore conditional execution is not available to the Thumb ISA. However, a way must be developed to represent the predicates of conditionalized instructions so that conditional execution feature can be restored. This paper borrows the IT (If-Then) instruction in Thumb-2 ISA which predicates the execution of up to four Thumb instructions [14]. The instructions affected by an IT instruction are said to be in an IT block.

A. Thumb-2 IT Instruction

The IT instruction takes the form:

IT{⟨x⟩{⟨y⟩{⟨z⟩}}} ⟨firstcond⟩

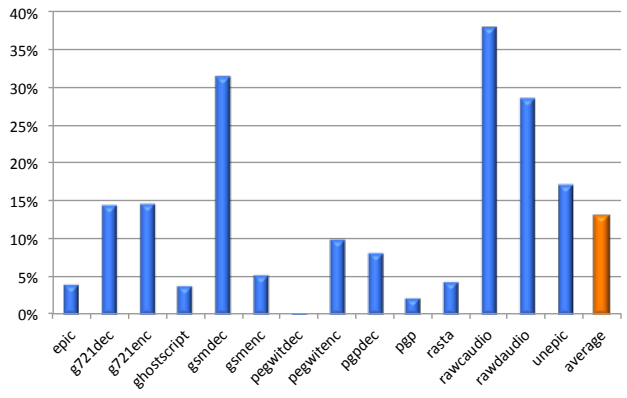
where x, y, and z are optional, and must be either T (for "then") or E (for "else"), while firstcond is the condition for the first instruction in the IT block. The encoding of 16-bit Thumb-2 IT instructions is shown in Figure 6(a).

B. 32-Bit IT Instruction

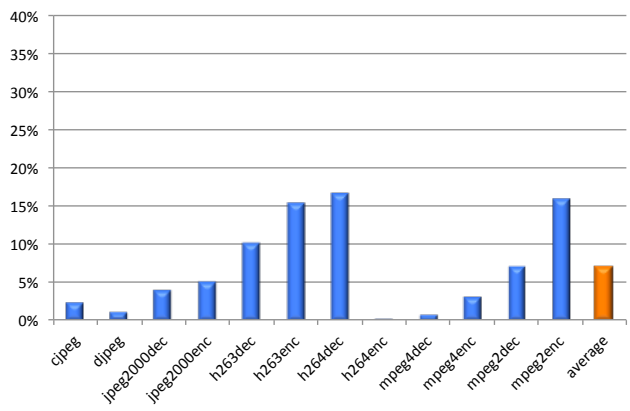
As the IT instruction is encoded by a 16-bit instruction format in Thumb-2 ISA, it needs to be extended into a 32-bit instruction format here. The most convenient way to encode a 32-Bit IT Instruction is to define it as an "architecturally undefined" instruction, and the instruction encoding for the new IT instruction is shown in Figure 6(b).

C. GCC Implementation

Modifying GCC to embed the 32-Bit IT Instruction in ARM code is quite straightforward, as GCC has already implemented if-conversion to generate predicated instructions. The only extra step that must be implemented in GCC is to convert conditionalized instructions into IT blocks preceded by IT instructions with the appropriate predicates. GCC just needs to look over every basic block and then transforms conditionalized instructions in the basic block into IT blocks. In addition, GNU assembler is updated to encode the 32-Bit IT Instruction into the new instruction format.



(a) MediaBench I



(b) MediaBench II

Figure 7. Ratios of Conditionalized Instructions

IV. Experimental Results

A. Setup

Experiments have been conducted by executing the MediaBench [12] and MediaBench II benchmarks [6] on SimpleScalar/ARM, a port of SimpleScalar [3] to ARM available from the University of Michigan. SimpleScalar/ARM simulates the five stage pipeline of ARM 9 and StrongARM processors, such as ARM9TDMI [1] and SA-1110 processor [9], which implements the ARM V4 architecture. Instructions are issued in-order by invoking the `sim-outorder` command with the option `-issue:inorder`. The configurations of L1 D-cache and I-cache for this processor are: 4KB cache size, 128B line size, 4-way associativity, and miss penalty of 1 cycle.

In addition to the base configuration of the original ARM, SimpleScalar/ARM has been modified to support the following configurations: ARM with 16/32 registers but no conditional execution, and ARM with 16/32 registers and the IT instruction. Specifically, there are five configurations:

Configuration	Registers	Conditional Execution	IT Instruction
base	16	Yes	No
reg16-noc	16	No	No
reg32-noc	32	No	No
reg16-it	16	No	Yes
reg32-it	32	No	Yes

B. Ratios of Conditionalized Instructions

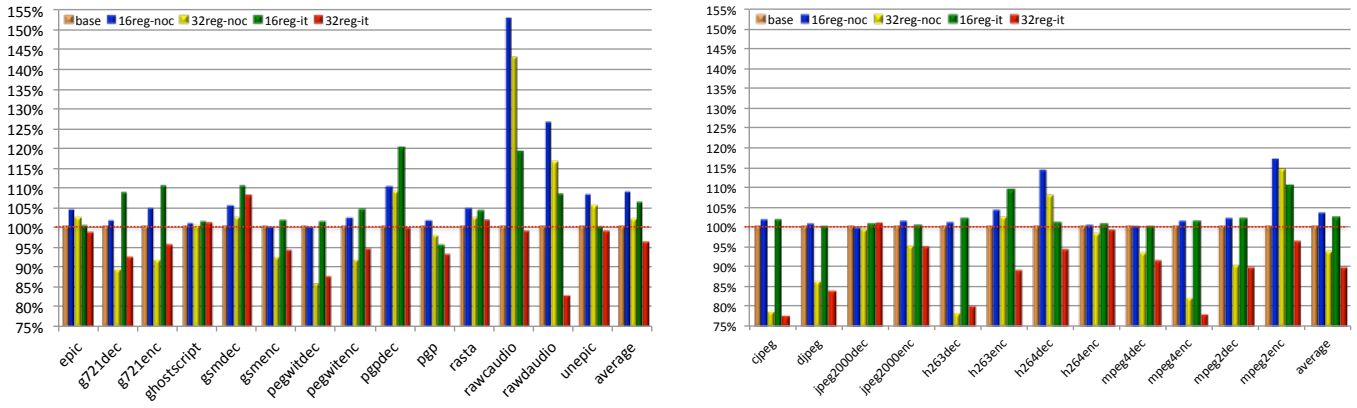
Since a condition field occupies a 4-bit space on every 32-bit ARM instruction, it is important to know if the condition fields of ARM instructions are underutilized. The more instructions that are conditionalized means the condition space is better utilized. Therefore, the ratio of conditionalized instructions executed by every benchmark program in MediaBench and MediaBench II has been measured by counting the number of instructions that are actually predicated. Figure 7(a) shows that 0.2%∼37.8% of instructions in MediaBench programs executed on ARM are conditionalized, and the overall average ratio of conditionalized instructions is about 13.0%. Similarly, Figure 7(b) reveals that only 0.3%∼15.8% of instructions in MediaBench II benchmarks executed on ARM are predicated, and the overall average ratio of predicated instructions is about 6.9%. This result indicates that the condition fields of most instruction are wasted for the MediaBench and MediaBench II benchmarks.

C. Normalized Execution Times

Figure 8 illustrates the normalized execution times of the MediaBench and MediaBench II benchmarks for different ARM configurations. The measured cycle counts of the MediaBench and MediaBench II benchmarks on these configurations have been normalized according to the base configuration.

Figure 8(a) shows that simply trading conditional execution for more registers is not beneficial for Mediabench benchmarks, which observe performance degradation of 9.0% and 2.3% on average respectively when the conditional execution feature is turned off on 16-register and 32-register ARM processors. The benchmark program rawcaudio performs the worst, suffering slowdowns 52.8% and 42.7% on reg16-noc and reg32-noc. respectively. The main reason is that there are 37.8% instructions can be conditionalized when the program is executed on the base ARM processor with conditional execution. Similarly, rawdaudio endures slowdowns of 26.6% and 16.4% on reg16-noc and reg32-noc respectively, as the ratio of conditionalized instructions is 28.5% on the original ARM. The performance gain obtained by doubling the number of registers is not enough to compensate the performance loss incurred by disabling the feature of conditional execution for the majority of the MediaBench benchmark programs.

Reactivating conditional execution using the IT instruc-



(a) MediaBench I

(b) MediaBench II

Figure 8. Normalized Execution Times

tion slightly improves the performance degradation, but the average runtime of MediaBench on a 16-register ARM is still worse than that of the original ARM by 6.2%. The reason of the performance discrepancy between the base and reg16-it configurations is that instructions on base can be conditionalized while extra IT instructions must be executed on reg16-it in order to handle conditional execution. Fortunately, doubling the number of registers usually can reverse the performance degradation. Runtime reduction can be observed on reg32-it for the most of the MediaBench programs, and the average performance improvement is 3.7%.

Figure 8(b) shows that performance would be slightly worse than the original ARM when conditional execution feature is turned off for the most of MediaBench II benchmarks, with the average slowdown of 3.6%. However, speedup can be observed for the most of MediaBench II benchmarks when the number of registers is doubled. The only three exceptions are *h263enc*, *h264dec*, and *mpeg2enc*, whose slowdown ratios are 2.6%, 8.0%, and 14.5%, respectively. The best speedup is achieved by *cjpeg*, which is 21.1%. The average performance improvement is 6.0% for MediaBench II benchmarks when the number of ISA registers is extended from 16 to 32. In other words, doubling the number of registers can achieve speedup for most of the MediaBench II benchmark programs even when the feature of conditional execution is disabled.

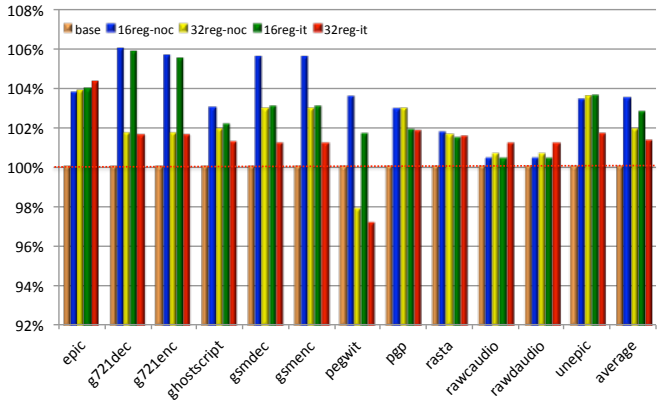
A 32-register ARM with conditional execution predicated by the IT instruction is the best configuration. Speedup can be observed for all MediaBench II benchmark programs. The only exception is the *jpeg2000dec* program, which is insensitive to conditional execution and shows no significant speedup or slowdown on all configurations. The best speedup reg32-it is achieved by *cjpeg*, which is 22.2% and the average performance improvement is 10.4% for MediaBench II benchmarks.

Table I
SPILLS IN OBJECT FILES

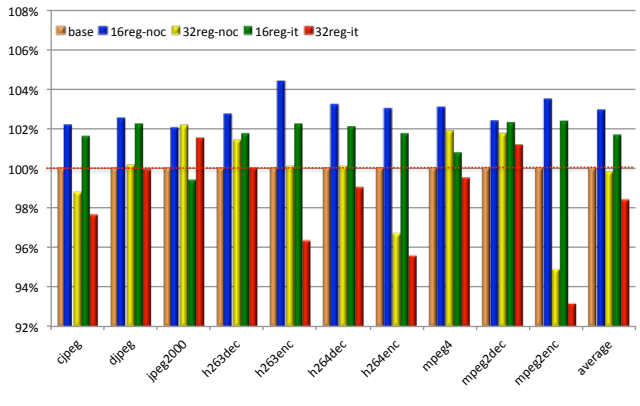
Program	16 Registers	32 Registers	Spill Reduction
MediaBench			
epic	2410	214	91.1%
g721dec	101	7	93.1%
g721enc	103	7	93.2%
ghostscript	9688	995	89.7%
gsm	595	178	70.1%
pegwit	739	8	98.9%
pgp	1248	1183	5.2%
rasta	3898	1114	71.4%
adpcm	14	0	100.0%
unepic	2399	214	91.1%
MediaBench II			
cjpeg	3285	502	84.7%
djpeg	3282	502	84.7%
jpeg2000	4902	520	89.4%
h263dec	9624	1492	84.5%
h263enc	1406	1060	24.6%
h264dec	104876	12757	87.8%
h264enc	49136	19473	60.4%
mpeg4	964	748	22.4%
mpeg2dec	571	18	96.8%
mpeg2enc	1753	144	91.8%

D. Register Spills

The performance improvement comes from the fact that register spills have been significantly reduced as more values have been stored in the extended register file. Table I tabulates the numbers of register spills that have been introduced by GCC into the target programs of the MediaBench and MediaBench II benchmarks. Note that the encoder and decoder of several benchmarks share the same object file. For instance, the JPEG 2000 encoder and decoder are stored in the same object program *jpeg2000*, and the MPEG 4 encoder and decoder are combined in the same file *mpeg4*. As the number of registers is extended from 16 to 32, the numbers of register spills are reduced drastically. More than

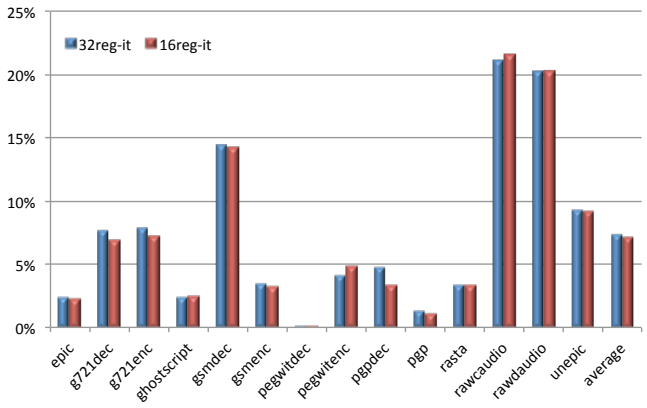


(a) MediaBench I

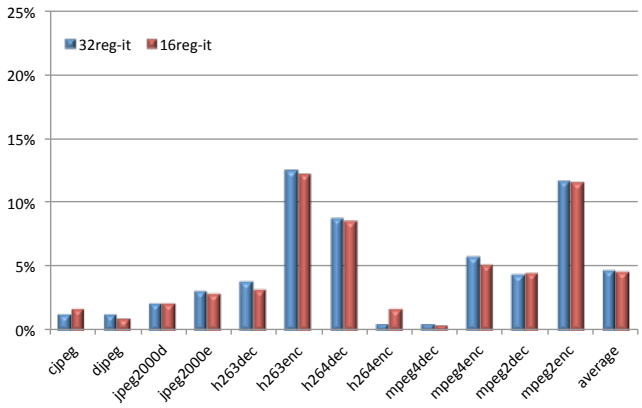


(b) MediaBench II

Figure 9. Normalized Code Sizes



(a) MediaBench I



(b) MediaBench II

Figure 10. IT Percentages

half of benchmark programs have observed reduction rates of over 80%. The average ratios of register spill reduction are 80.4% and 72.7% for MediaBench and MediaBench II, respectively.

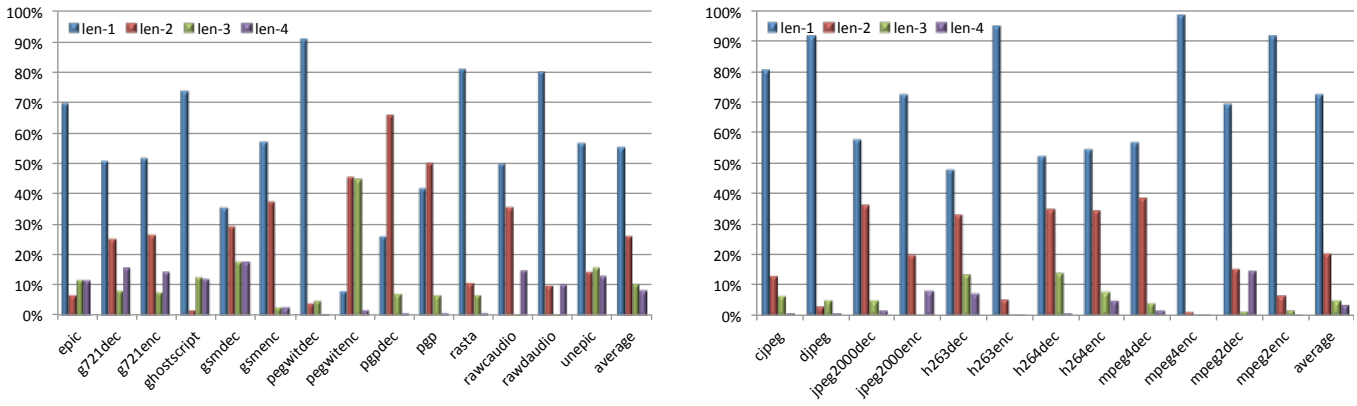
E. Normalized Code Sizes

As the length of every instruction remains the same, the code sizes of generated object files will not change much. The difference is caused by replacing conditionalized instructions with conditional branch instructions or IT instructions, and by reducing register spill instructions when the register file is doubled. Figure 9 depicts the normalized code sizes of the MediaBench and MediaBench II benchmarks for different ARM configurations. Figure 9(a) shows that the average code size expansion of MediaBench is 3.5% when the conditional execution feature is turned off, and 1.9% when the register file is doubled. If conditionalized instructions are replaced by IT instructions, the average code size expansion of MediaBench binaries is 2.8% on the 16-register ARM, and 1.3% on the 32-register ARM.

Figure 9(b) shows that the average code size of MediaBench II benchmarks is expanded by 2.9% when the conditional execution feature is turned off and reduced by 0.2% when the size of the register file is doubled. If conditionalized instructions are replaced by IT instructions, the average code size of MediaBench binaries is increased by 1.6% on the 16-register ARM, and decreased by 1.6% on the 32-register ARM.

F. IT Instructions

As IT instructions are used to replace conditionalized instructions, the numbers of IT instructions in the binaries for the new ARM configurations should be proportional to the amounts of conditionalized instructions in the object code for the base ARM. Figure 10 displays the ratios of IT instructions to the total instruction counts of MediaBench and MediaBench II benchmarks executed on the 32-register and 16-register ARM processors. When comparing with



(a) MediaBench I

(b) MediaBench II

Figure 11. Breakdown of it Instructions

the numbers in Figure 7, the ratio of IT instructions is smaller than the ratios of conditionalized instructions for every MediaBench and MediaBench II benchmark, as each IT instruction represents the predicates of up to four instructions.

Figure 11 depicts the breakdown of the numbers of instructions that are conditionalized by every IT instruction in MediaBench and MediaBench II benchmarks on a 32-register ARM. It reveals that the IT instructions generated by GCC are not efficient, namely, the major of IT instructions control only one instruction each. If GCC can generate more efficient IT instructions, further performance gain should be observed.

V. Related Work

Register Connection is a scheme to make more registers accessible to the compiler by mapping architectural registers to physical registers dynamically [10]. Special instructions are added to designate the mapping and there is a mapping table managed by the hardware. The additional code size introduced by these instructions is significant.

Krishnaswamy and Gupta have proposed an approach to use the invisible registers in Thumb code, as only half of the register file is visible to most Thumb instructions [11]. A special instruction is introduced to change the visible subset of registers at any program point and hence the compiler can make use of all registers in all instructions through the use of this instruction. A register allocation algorithm has been developed to allocate invisible registers and embed the special instructions.

Zhuang et. al. have developed a hardware managed register allocation scheme to allocate more physical registers at runtime, i.e. managing extended registers not addressable through ISA with hardware support [17]. Their approach is complicated but the performance improvement is only moderate.

Differential Register Allocation can also double the number of registers, but the mechanism is different [16]. It encodes the register field using the difference between consecutive register accesses to allow the compiler to allocate more registers than can be specified using a regular encoding, while this work uses the underutilized condition bits to double the number of architectural registers. Therefore, this approach can be used in conjunction with this work to address more registers.

VI. Conclusions and Future Work

This paper proposed an approach to double the number of architectural registers on ARM while maintaining the instruction width. It first freed up the underutilized condition field in the ARM instruction encoding by disabling conditional execution feature, and then used the space to accommodate the extra bits that were need for register fields when the number of registers was doubled. It then restored the feature of conditional execution by borrowing the IT instruction from Thumb II ISA to represent the predicates of conditionalized instructions on the 32-register ARM. Experimental results showed that the performance could be improved by 10.4% on average for MediaBench II benchmarks on ARM processors with 16 extra ISA registers.

There are two on-going projects that try to address the issue of backward compatibility using different approaches. One project is developing a binary rewriting tool that transforms ARM instructions to the new formats and performs register reallocation to utilize the extra registers. The other project is trying to enable GNU compiler to decide when it is beneficial to generate ARM code with 32 registers and when it is profitable to retain the conditional execution feature.

Acknowledgment

This research was supported in part by MOEA project 98-EC-17-A-01-S1-034 and NSC grant NSC98-2221-E-011-065-MY3.

REFERENCES

[1] ARM Limited. *ARM9TDMITM Technical Reference Manual*, 2000.

[2] Todd Austin, David Blaauw, Scott Mahlke, Trevor Mudge, Chaitali Chakrabarti, and Wayne Wolf. Mobile supercomputers. *IEEE Computer*, 37(5):81–83, May 2004.

[3] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, 2002.

[4] Huang-Jia Cheng and Yuan-Shin Hwang. Trading conditional execution for more registers on ARM. In *Proceedings of the 2010 EUC*, pages 53–59, 2010.

[5] Edil S. T. Fernandes, Anna Dolejsi Santos, and Claudio L. de Amorim. Conditional execution: An approach for eliminating the basic block barriers. *Microprocessing and Microprogrammin*, 40:689–692, 1994.

[6] Jason Fritts and Bill Mangione-Smith. MediaBench II - technology, status, and cooperation. In *Proceedings of the Workshop on Media and Stream Processors*, 2002.

[7] GCC. The GNU compiler collection. http://gcc.gnu.org/.

[8] Wen-Mei Hwu. Technology outlook: Introduction to predicated execution. *IEEE Computer*, 31(1):49–50, January 1998.

[9] Intel Corporation. *Intel® StrongARM SA-1110 Microprocessor Developer's Manual*, October 2001.

[10] Tokuzo Kiyohara, Scott Mahlke, William Chen, Roger Bringmann, Richard Hank, Sadun Anik, and Wen-Mei Hwu. Register connection: a new approach to adding registers into instruction set architectures. In *Proceedings of the 20th annual international symposium on Computer architecture*, pages 247–256, 1993.

[11] Arvind Krishnaswamy and Rajiv Gupta. Efficient use of invisible registers in thumb code. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*, 2005.

[12] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-smith. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'97)*, pages 330–335, 1997.

[13] Joseph C. H. Park and Mike Schlansker. On predicated execution. Technical Report HPL-91-58, HP Labs, 1991.

[14] David Seal, editor. *ARM Architecture Reference Manual*. Addison-Wesley Professional, 2nd edition, 2001.

[15] Simon Segars. Low power design techniques for microprocessors. In *2001 IEEE International Solid-State Circuits Conference (ISSCC)*, 2001.

[16] Xiaotong Zhuang and Santosh Pande. Differential register allocation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI'05)*, pages 168–179, 2005.

[17] Xiaotong Zhuang, Tao Zhang, and Santosh Pande. Hardware-managed register allocation for embedded processors. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 192–201, 2004.