Improving CISC Instruction Decoding Performance Using a Fill Unit

Mark Smotherman

Dept. of Computer Science
Clemson University
Clemson, SC 29634-1906
mark@cs.clemson.edu

Manoj Franklin

Dept. of Elect. and Computer Eng.
Clemson University
Clemson, SC 29634-0915
mfrankl@eng.clemson.edu

Abstract

Current superscalar processors, both RISC and CISC, require substantial instruction fetch and decode bandwidth to keep multiple functional units utilized. While CISC instructions can sometimes provide reduced fetch bandwidth requirements, they are correspondingly more difficult to decode. A hardware assist, called a fill unit, can dynamically collect decoded microoperations into a decoded instruction cache. Future code fetches to those locations can be satisfied out of this cache and thus bypass the decoding logic. This approach is investigated using the Intel x86 architecture, and a speedup of approximately a factor of two over a P6-like decoding structure is seen for the three SPEC benchmarks investigated. This design is accompanied by a microengine-register allocation and renaming scheme that prevents the increased supply of microoperations from placing excessive demands on the normal register renaming hardware.

1 Introduction

Modern superscalar processors like the HP PA 8000, IBM/Motorola PowerPC 604, Intel P6, and MIPS R10000 are built around a dynamically scheduled microengine. This style of design has often been referred to as restricted data flow [8]. The microengine typically contains many functional units and has the ability to contain tens of instructions in various stages of execution in an out-of-order manner. While most of the aforementioned processors have instruction sets classified as RISC, the Intel P6 ([13]) and earlier work by Patt and colleagues on a VAX design ([10], [12]) demonstrate that the advantages of restricted dataflow can apply to CISC instruction sets also.

A microengine with multiple functional units needs a steady supply of instructions to best utilize its hardware. Some have proposed multithreaded organizations to provide a large pool of guaranteed-independent instructions, while others have investigated VLIW and wide-issue superscalars. Many studies assume an easily-decodable RISC instruction format and scale up the decoding hardware as needed. However, the limitations on decoding in the Intel P6 perhaps give a hint that CISC decoding does not scale as well as might be desired [13].

Patt and his colleagues have been investigating dynamically scheduled microengines over the past ten years, beginning with the HPS in 1985 [6] [8] [9]. From the beginning they considered that a decoded instruction cache might be a useful supplement to the microengine to assist performance. In 1988, Melvin, Shebanow, and Patt described in more detail a hardware assist, called a fill unit, to compact microoperations generated from sequentially-fetched instructions into a decoded instruction cache [7]. The purpose of the fill unit was to construct a larger piece of atomic work that could be given to the dynamic scheduler and thereby increase the utilization of the functional units. In their design, architected registers (i.e., those specified in the instruction set architecture) as well as microengine registers (i.e., logical registers visible at the microarchitecture level only) were renamed to encode forwarding requirements between and within instructions. Microoperation references were also renamed into the address space of the decoded instruction cache, and microoperations from a single instruction could be split across two lines in the decoded instruction cache. Filling stopped (i.e., a decoded instruction cache line was "finalized") whenever a branch was encountered or no empty microoperation slots remained in the filled line. Microoperations with data dependencies would be executed in the proper order by the underlying dynamically scheduling hardware.

Patt and colleagues also proposed a decoded instruction cache ("node cache") to assist the performance of an HPS version of the DEC VAX [10] [12]. The VAX is a CISC and has variable-length instruction formats; an average VAX instruction generates about

four HPS microoperations. The node cache assisted in keeping the decoding rate at one VAX instruction per cycle, and simulations indicated that the average CPI of 6 observed in contemporaneous VAX implementations could be reduced to 2 in the HPS version. This was remarkable since the VAX instruction set includes many data-dependent operations that cannot be easily cached as fixed sequences of microoperations (e.g., the procedure return instruction provides automatic register restoring but this is dependent on a register save mask located in the procedure's stack frame). Across a set of small benchmarks, including daxpy and Dhrystone, between 60% and 100% of the instructions could be stored as microoperations in the node cache. A decoded instruction cache was also part of subsequent work on an extended Motorola 88110 (RISC) design [2].

In 1994 Franklin and Smotherman investigated a fill unit accelerator for a simple pipelined implementation that grouped RISC instructions into a decoded instruction cache for later multiple issue [4]. With a simple statically-scheduled pipeline and no renaming, they found that high performance required cascaded (or "fused") functional units to handle dependent operations and a tree-like line format to provide instructions from both paths past a conditional branch. This paper represents an extension of that work to a CISC instruction set for which a dynamically scheduled microengine handles the dependent operations, much like the HPS-VAX.

This paper is organized as six sections. The introduction has reviewed the fill unit approach to decoded instruction caches. Section 2 reviews the decoding structure of a P6-like processor, and Section 3 describes how a fill unit could be applied to enhance the decoder performance in such a design. Section 4 investigates the register renaming requirements of the two approaches and presents a new method of renaming microarchitected registers, which is ideally suited for a decoded instruction cache. Section 5 presents results from the simulation of instruction traces of three SPEC benchmarks; and, a summary and conclusions appear in section 6.

2 Decoding instructions in a P6-like manner

A decoding structure similar to what has been publicly stated about the Intel P6 design [13] is depicted in Figure 1. Up to three instructions can be decoded simultaneously, with the restriction that only the first instruction in the instruction queue can produce multiple microoperations (i.e., it can be a *complex* instruction). The next two instructions are decoded only if they will produce one microoperation each (i.e., they are *simple* instructions, such as register-to-register add). If a complex instruction appears in the second or third decode slot, it must wait until it reaches the first decode slot, to be decoded there by the complex instruction decoder. This may result in instances of one-decode or two-decodes per cycle rather than the optimal three-decodes per cycle. For example, an instruction sequence of simple, complex, simple, complex requires three cycles to completely decode; this is because only the initial simple instruction can be decoded on the first cycle, the complex/simple pair can be decoded on the second cycle, and the last complex instruction cannot start decoding until the third cycle.

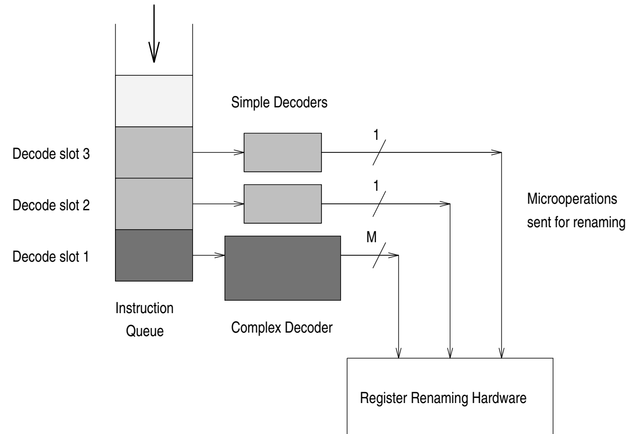

Figure 1: P6-like decoder.

To lay the foundation for our solution to remove some of the limitations of P6-like decoding structures, let us first study the resource requirements of complex as well as simple instructions. (Since the details of the P6 have not been made public, we are assuming a fairly simple model to obtain rough estimates of the decoding performance of its apparent approach to distinguishing between complex and simple instructions; the P6 will likely operate in a different manner.) We consider as complex those instructions that can be of the form memory-to-register or register-to-memory, and consider as simple those instructions that are of the form register-to-register. The complex instruction form we are postulating allows not only a load from memory to fetch one of the operands, but it also allows the result to be written back to memory.

Consider the complex register-to-memory instruction `ADD [EBX+EAX],ECX`. This instruction can be split into the following four microoperations. (We are ignoring the use of segment registers to form a physical memory address.)

```
MR1 <- EBX + EAX          ; eff. addr. calc.
MR2 <- Mem[MR1]           ; operand load
Flags, MR3 <- MR2 + ECX   ; add
Mem[MR1] <- MR3           ; result store
```

Figure 2 shows the data flow between these microoperations. MR1 provides the address value from the effective address calculation to both the load and the store microoperations. MR2 provides the loaded value to the computation microoperation, and MR3 forwards the computed value to the store microop-

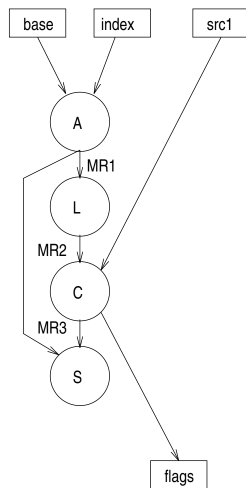

Figure 2: ADD [EBX+EAX],ECX mapped onto microoperations.

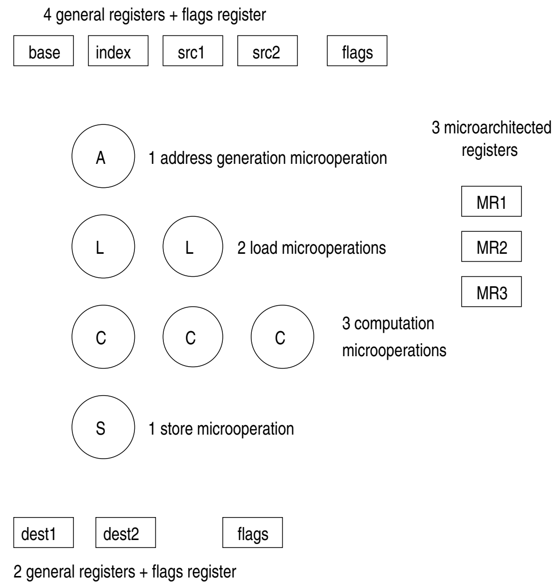

Figure 3: Microoperation resource template.

eration. Thus, for this instruction, three architected registers are needed for input (`EAX`,`EBX`,`ECX`), one architected register is used for output (`Flags`), and three microarchitected registers are used for intermediate values.

The appendix contains other examples of instructions mapped onto microoperations; taken together they provide the resource template shown in Figure 3. (In the P6 there are a few very complicated instruction forms that must be handled by a series of microoperations fetched by the complex decoder from a microcode store, e.g., `ENTER`, `POPA`, `PUSHA`; we are not modeling these in the current study.) The resource template has seven microoperations. Up to five architected registers, including the flags register, can be used as input (top of template) and up to three architected registers, including the flags register, can be written (bottom of template). In our model there are also up to three microengine registers (called MR1, MR2, and MR3) used to forward data within the complex instruction between the microoperations. Instructions shown and discussed in the appendix demonstrate that a maximum of four source registers are required; this is for a divide using a divisor fetched from memory. The two load and three computation microoperations shown in Figure 3 are required by the compare string instruction; and, the two destination registers and the flags register shown in the figure are required for both the compare string and the multiply instructions.

Next, let us see the resource requirements for a sequence of instructions that are decoded in a cycle. Figure 4 shows the microoperation-level resource template for a triple-decode cycle in the P6-like processor. The first template corresponds to that of a complex instruction and the subsequent two correspond to that of simple instructions. Consider the following sequence of complex, simple, simple instructions:

```
ADD   [EBX+EAX], ECX
ADD   EAX, 4
INC   ECX
```

These three instructions can be decoded simultaneously by a P6-like decoder and adhere to the resource template given in Figure 4.

To allow for out-of-order execution, all architected registers and microarchitected registers involved in the microoperations produced by the decoder must be renamed by the underlying microengine. This renaming must be done in sequence for the first instruction in a pair or triple and then the second (and then the third) to be able to resolve dependencies between the two (three) instructions. For example, the destination register of the first instruction can be one of the source registers for the second (or third) instruction. The formation of a physical memory address in the x86 also requires access to segment registers. We assume that these registers are static and are not renamed; rather, any change to a segment register forces the processor to serialize.

3 Collecting complex microoperation sequences with a fill unit

As described in the introduction, we propose to use a fill unit to collect decoded microoperations and then store them into the lines of a decoded instruction cache.

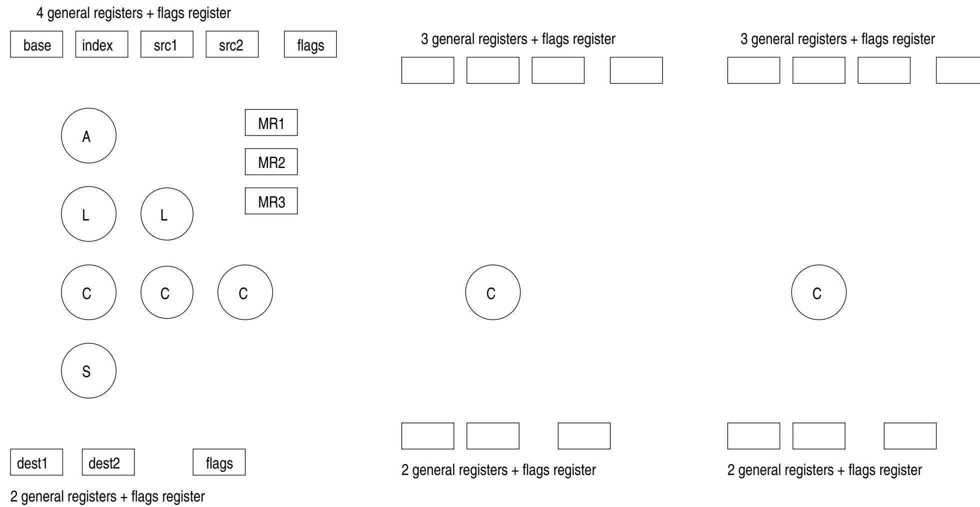

Figure 4: Simultaneous, triple-instruction-decode microoperations using P6-like decoder.

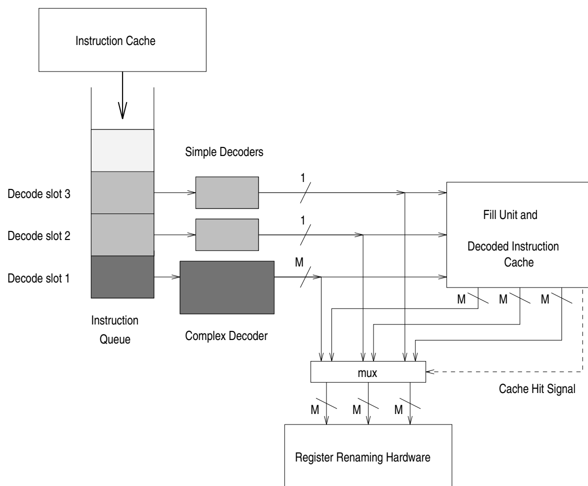

Figure 5: Fill unit decoding.

3.1 General operation

Figure 5 depicts the fill unit approach. If the instruction fetch address matches that of a decoded instruction cache entry, the decoded instruction cache is given priority over the normal instruction cache and decoding path, and decoded microoperations are supplied directly to the register renaming stage from the decoded instruction cache. Note that by storing decoded microoperations in the cache, there is no need to repeat the fetching of instructions from the normal instruction cache nor decode them; however, register renaming must still occur prior to execution. The advantages of the fill unit are that it can collect the decoded microoperations of more than one complex instruction together into the same cache line and that a complex instruction can appear in some other position than what would be the first decode slot.

Because the decoded microoperations from a variable number of instructions can be supplied by a particular decoded instruction cache line, each line must also include a next-address field and supply it to the branch unit. (Note that a program-counter-increment value is less desirable since the required address addition could delay the start of the next instruction fetch.) Filled lines are stored in the decoded instruction cache only if they represent more than one instruction, as it is more storage efficient to fetch single instructions directly from the normal instruction cache.

An exception that occurs while executing a group of decoded microoperations from a decoded instruction cache line can be handled by the mechanisms already provided for the dynamic scheduled microengine; for example, the processor can flush the group of microoperations, enter a scalar-only execution mode, and reexecute the instructions in the group to the point of the exception. In the exception-handling mode, the fill unit can be disabled.

The decoded instruction cache line could potentially include as many microoperations as possible, with potential line crossings by microoperations (similar to [7]), but for this study we will constrain it to the 21 microoperations possible from three complex instructions (i.e., from replicating Figure 3 three times), and require that decoded instruction cache lines end on instruction boundaries. This approach will simplify exception handling and limit the additional renaming hardware needed to provide correct sequencing between microoperations in different decoded instruction cache lines.

3.2 Data and control dependencies

Data dependencies and resource collisions will not prevent filling of the microoperations but will instead be resolved by the register renaming hardware and the underlying dynamically scheduled microengine. The microarchitected registers are used to forward values between the microoperations.

For a basic design, control dependencies such as branches will stop the filling of a decoded instruction cache line; this can cause the average number of microoperations placed into a decoded instruction cache line to be less than the ideal. A next-instruction-address field is used to provide correct instruction sequencing. If there is no branch or control-changing instruction in a decoded instruction cache line, this field is set to the address of the first instruction that was not filled into the line. A branch is represented in the decoded instruction cache line by two separate fields: a condition and a branch address. When the last instruction to be filled is a conditional branch, the next-instruction-address field serves as the branch-untaken address and the branch-address field serves as the branch-taken address.

When the last instruction to be filled is an unconditional branch, the branch-address field is unused and the branch target address is placed in the next-instruction-address field. Branching to an instruction within a previously filled line presents no logical problems [4]. Since the decoded instruction cache has no line with that starting address, the fetch will be made from the normal instruction cache, and a new fill line will start at the branch target.

As in previous work with the fill unit ([4]), experiments with a design that finalized cache lines on a branch saw limited speedup in the of instruction decoding rate. The approach that we have taken is to retain one branch per decoded instruction cache line but to widen the line to include the sets of microoperations from instructions along both the untaken path and the taken path, similar to a VLIW tree instruction [1] [3]. Only one set of instructions will be sent to the renaming stage, and the selection will be based on a prediction returned from a branch prediction cache or a default heuristic. The underlying rule is that the microoperations from instructions prior to the branch, if any, must appear on both paths.

In the normal filling of a decoded instruction cache line, both parts of the line are filled with the same microoperations up to a branch. If possible, then the branch is filled into a single branch microoperation field. The path not followed has its next address field set (i.e., the branch target address in case of an untaken branch or the address beyond the branch instruction for a taken branch), while the fill unit attempts to continue to fill microoperations along the path followed by the current execution. The branch prediction cache is updated with a taken/untaken predictor to control which set of microoperations will be renamed and issued.

If a misprediction has been made, the wrong set of microoperations will have been sent to the renaming stage, so those microoperations are flushed, the branch prediction cache is updated, and the decoded instruction cache line is refetched but with the alternate path. Since the alternate path may not have been traversed before, the set of microoperations for this path may only include the microoperations prior to the branch, if any, and the branch. Thus, the fill unit can only provide microoperations up to and including the branch. However if the next instruction fetch occurs from the normal instruction cache, the fill unit can back up and attempt to continue filling the tree-like line with microoperations from the new path.

Self-modifying code can be supported by invalidating a range of addresses [5], since we will not know in exactly which decoded instruction cache line a stale instruction is held. A bound of 16 bytes per instruction and three instructions per decoded instruction cache line yields a range of 48 bytes. The invalidation mechanism can drop the last four bits of the tag compare and send four invalidates using increment-by-16 addresses. Note that the tree-like line must keep two tags for this comparison: one for the entry address and one for the target address of any target instructions filled into the line.

4 Register renaming requirements

As mentioned in Section 2, decoded instructions must be sent through the register renaming hardware to allow the dynamic scheduling hardware to fully exploit the instruction level parallelism available in the instruction window. Figure 6 illustrates the renaming that happens in P6-like hardware. Renaming is done for the architected source and destination registers, as well as the microarchitected registers that forward results between individual microoperations. Note that the architected and microarchitected registers must be renamed in a manner that preserves internal dependencies from instruction to instruction and microoperation to microoperation.

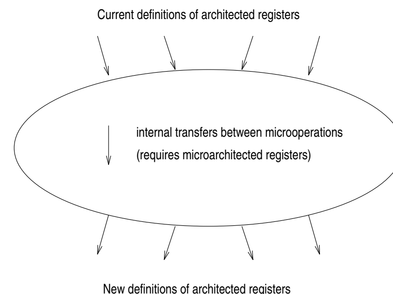

Figure 6: Renaming requirements for multiple microoperation groupings.

In the P6-like scheme we have outlined there can be a theoretical maximum of 13 architected source registers, nine architected destination registers, and three microarchitected registers that need to be renamed in a cycle. In the fill unit case there can be a theoreti-

cal maximum of 15 architected source registers, nine architected destination registers, and nine microarchitected registers to be renamed in a cycle. The actual renaming required is much less than the theoretical, as shown later in the experimental results section.

On first glance, it may appear that the fill unit approach places an excessive burden on the register renaming hardware's peak bandwidth, since the decoded instruction cache can supply multiple complex instructions per cycle. However, on the contrary, the fill unit approach can be used to reduce the peak demands on the register renaming hardware (compared to the case with no fill unit). This can be accomplished by providing the microarchitected registers required by a decoded instruction cache line in a separate physical register file, and letting the fill unit do separate renaming for those fill unit registers.

Figure 7 illustrates how this scheme works. When a line is fetched from the decoded instruction cache, it is sent through special fill unit renaming hardware, which renames the microarchitected registers (possibly up to nine) to names in the separate fill unit register file. These microoperations, now in need of renaming only for the architected registers, are sent through the general renaming hardware. When decoded microoperations are supplied by the decoder, all architected and any remaining microarchitected registers are sent to the general renaming hardware (as before), which renames them to physical register names.

One simple way of doing the fill unit renaming is to append a unique identifier to all references to microarchitected registers. This identifier can be constructed by using the bits of an access counter of decoded instruction cache lines. Since each decoded instruction cache line must have at least two instructions worth of microoperations and since there is a limit of, say, fifty instructions in the reorder buffer of the microengine, there will be a limit of at most 25 decoded instruction cache lines present in the microengine at one time. Thus a five-bit counter would suffice to prevent collisions among microarchitected registers from decoded instruction cache lines, and combined with the four bits needed for the nine possible microarchitected registers per decoded instruction cache line yields a fill unit register file of size 512. This special renaming by bit pattern insertion is simple and fast, and thus would not require another pipeline stage or unduly lengthen cycle time.

We can also reduce the number of renamings by treating all microoperations from a decoded instruction cache line as an atomic unit at the instruction level [7]. That is, we can retire them as a single unit or not; if we cannot retire them as a unit due to, say, an exception caused by one of the included microoperations, then we must back up and do the normal scalar fetch, issue, execute, and retire up to the point of the exception. However, treating the grouped microoperations as a single atomic unit has the advantage that multiple updates of any given architected register are reduced to only one. This is especially helpful for an often-written register like the `Flags` register on the x86. As an example, consider

```
ADD   EAX, [EBX+ESI]
ADD   EAX, 10
ADD   EBX, 4
```

This instruction sequence requires the following register transfers (ignoring segment registers)

```
MR1         <- EBX + ESI
MR2         <- memory[MR1]
EAX, Flags <- EAX + MR2
EAX, Flags <- EAX + 10
EBX, Flags <- EBX + 4
```

so the multiple-issue of these microoperations by a P6-like decoder requires the following renaming actions (which must be done in sequence to preserve the dependencies)

```
use EBX, ESI   def MR1
use MR1        def MR2
use EAX, MR2   def EAX, Flags
use EAX        def EAX, Flags
use EBX        def EBX, Flags
```

Our scheme for using special fill unit registers reduces this to

```
use EAX, EBX, ESI  def EAX, Flags
use EAX            def EAX, Flags
use EBX            def EBX, Flags
```

But if we treat this sequence of three instructions as an atomic unit with single definitions (all but the last definition per register are dropped) the registers transfers then become

```
MR1                  <- EBX + ESI
MR2                  <- memory[MR1]
MR3, MR4             <- EAX + MR2
EAX/MR5, MR6         <- MR3 + 10
MR7/EBX, MR8/Flags <- EBX + 4
```

with renaming requirements

```
use EBX, ESI, EAX, EBX  def EAX, EBX, Flags
```

So, we have reduced the renaming requirements from 15 registers (11 architected registers and 4 microarchitected registers) to an atomic issue using 7 architected registers. The tradeoff is that we now need a maximum of 15 special fill unit registers: three each for internal transfers within the three complex instructions, three for forwarding between the first instruction and second and third, and three for forwarding between the second instruction and third. Note that 15 fits within the four bits needed for specifying 9 internal transfer registers, so the separate register file need not be enlarged.

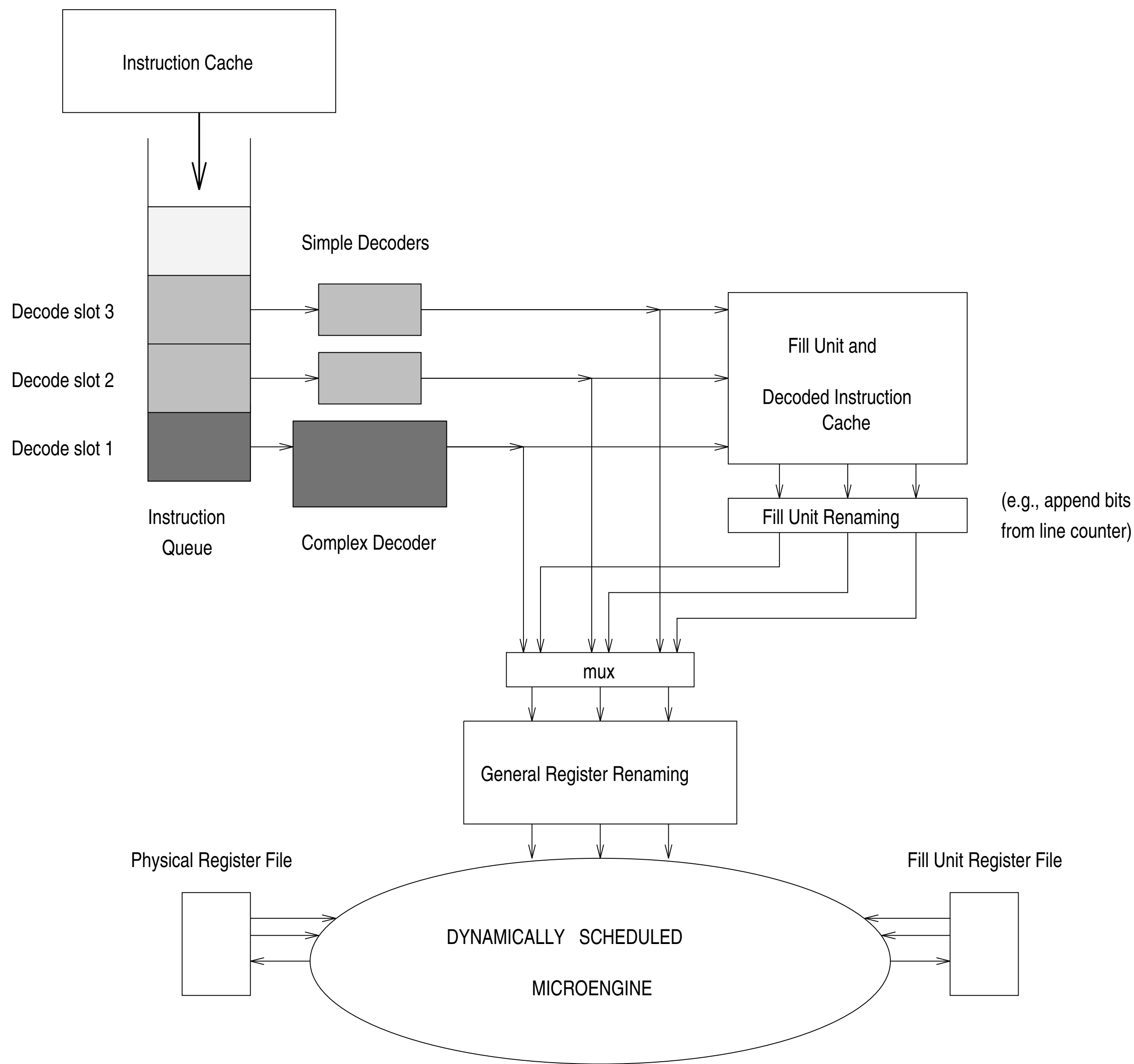

Figure 7: Fill unit renaming.

5 Experimental results

All data reported in this paper were gathered with a simulator that accepts x86 instruction traces in the kernel trace format (produced by the IDT program [11]) and simulates instruction decoding, keeping track of relevant decoding and renaming information on a cycle-by-cycle basis. Execution latencies are not modeled, and the normal instruction cache is assumed to have infinite size with single-cycle fetch. For benchmarks, we use three integer programs from the SPEC92 suite compiled by the Metaware compiler with optimization flag -O. Data is presented for compress with input file in, espresso with input file opa.in, and xlisp with input file li-input.lsp, which are integer-intensive programs written in C. The instruction mix of the three programs is given in Table 1.

Instruction type	*compress*	*espresso*	*xlisp*
Register-to-register	28%	30%	27%
Move (r-m, m-r)	27%	24%	21%
Complex	26%	20%	28%
Branch	19%	26%	24%

Table 1: Benchmark program characteristics.

5.1 Instruction decoding rate

Table 2 shows the results of simulating the three benchmark programs using the P6-like decoder. Notice that although the decoder can decode up to three instructions per cycle, it is able to achieve an average decoding rate of only slightly over 1.3 instructions per cycle when only register-to-register operations are considered as simple and allowed to decode in parallel (first row). This severely restricts the dynamic scheduling hardware from exploiting the instruction level parallelism that may be present. If we extend the definition of what simple decoders can handle to include the register-to-memory and memory-to-register move operations (i.e., essentially RISC-like loads and stores), the decoding rate increases to over 1.7 (second row). Finally, if we consider a scheme with three complex decoders allowed to decode in parallel, the instruction stream is limited only by taken branches and the rate is close to three instructions decoded per cycle (third row).

Table 3 presents the results of simulating a basic fill unit with a 1K-entry, 4-way set associative decoded instruction cache. Let us study these results in some detail. When a basic fill unit is used, the instruction decoding rate has substantially improved to 2.0-2.2, providing an improvement of 40-65% over the case where no fill unit is used. Increasing the instruction decoding rate essentially increases the supply of microoperations to the dynamic scheduling hardware, providing it more opportunities to look for and exploit instruction level parallelism. Table 3 also lists other metrics, such as the hit rate of the decoded instruction cache (i.e., fraction of times the decoded instruction cache supplied a line when it was queried), the average size of decoded instruction cache lines, and the fraction of times the fill unit finalized a line due to the occurrence of a branch instruction. It is important to note that branches account for the major reason for finalization of lines.

Table 4 presents the results obtained with a fill unit that generates tree-like decoded instruction cache lines. These results demonstrate the value of filling both paths past a branch. With tree-like lines, the instruction decoding rate has dramatically increased to 2.6-2.9. This corresponds to an 85-125% increase over the original case (i.e., without the fill unit). The decoded instruction cache hit rate has also increased substantially, as more lines are now filled by the fill unit. Notice that there is not much increase in the average size of a line; this is because the average is not taken over the same sample. When tree-like lines are used, many more lines are formed, as indicated by the increase in the decoded instruction cache hit rate.

Figure 8 shows that the decoding rate per cycle increases from approximately 1.5 instructions per cycle for a 32-entry cache to approximately 2.75 for a 16K-entry cache. (Note each cache line is likely to be 50-100 bytes in length.)

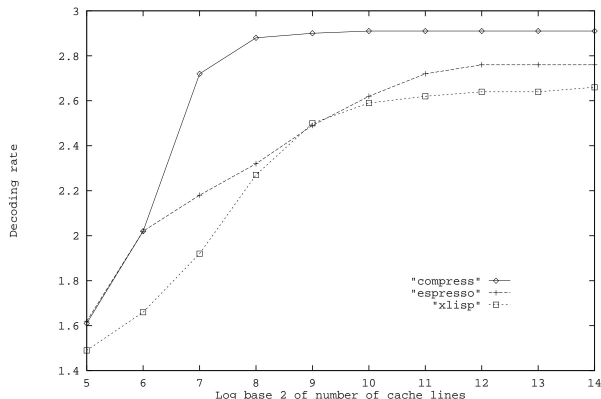

Figure 8: Decoding rate as a function of decoded instruction cache size.

5.2 Register renaming requirements

Now let us look at the register renaming requirements. It will help little to substantially increase the bandwidth of the instruction decoding if we merely cause a bottleneck elsewhere. Table 5 presents the results of renaming for a 1K-entry decoded instruction cache. If fill unit renaming is not done, then the average number of renames per cycle would triple from 5 to almost 15. With fill unit renaming, the average is less than double, 5 to 8.

Instructions decoded per cycle	*compress*	*espresso*	*xlisp*
Register-to-register as simple	1.30	1.41	1.33
R-R and moves as simple	1.74	1.90	1.71
Three complex decoders	2.82	2.74	2.77

Table 2: Instruction decoding rate without fill unit.

Metric	*compress*	*espresso*	*xlisp*
Instruction decoding rate with fill unit	2.15	1.98	2.18
Hit ratio of decoded instruction cache	75.9%	56.8%	73.3%
Average size of decoded instruction cache line	2.9	2.8	2.8
Finalization due to return or indirect jump	6.3%	3.5%	5.0%
Finalization due to branch	26.8%	35.7%	23.6%

Table 3: Results with basic fill unit.

Metric	*compress*	*espresso*	*xlisp*
Instruction decode rating with tree-like line	2.91	2.62	2.59
Hit ratio of decoded instruction cache	99.9%	95.6%	90.3%
Average size of decoded instruction cache line	2.9	2.8	2.9
Finalization due to return or indirect jump	17.4%	20.3%	14.1%
Finalization due to branch	1.6%	0.9%	4.7%

Table 4: Results with tree-like line.

Type		*R-R as simple*	*R-R and moves as simple*	*With Fill Unit (Tree-Like Line)*
Architected registers	Reads	1.64	2.19	2.85
	Writes	1.26	1.68	2.30
Microarchitected registers	Reads	1.10	1.47	1.45
	Writes	1.01	1.35	1.41
Total general renaming		5.01	6.69	8.01
Fill unit registers	Reads			3.59
	Writes			2.98
Total fill unit renaming				6.57

Table 5: Average renaming frequencies.

6 Conclusions and future work

We have taken the idea of a fill unit, originally proposed by Melvin, Shebanow, and Patt, and applied it to the problem of increasing the instruction decoding rate for a CISC instruction stream. The results of our experiments with the x86 instruction set show that we can obtain a factor of two speedup in decoding performance compared with a P6-like decoding structure.

We also presented a microengine-register allocation and renaming scheme based on registers available only within a decoded instruction cache line. This scheme prevents the increased supply of microoperations from tripling the register renaming demand on average. Instead we were able to limit the increase of average demand on the register renaming hardware to less than a factor of two. We plan to further investigate the performance tradeoffs involved in renaming, such as the effects of a limited number of renaming ports on fill line finalization. Instruction selection and scheduling by the compiler will also have an impact on the decoding rate, and we intend to investigate the amount of performance improvement obtainable from the compiler.

Acknowledgements

We gratefully acknowledge the encouragement by Yale Patt during this work and the provision of x86 instruction traces by Chih-Chieh Lee of the University of Michigan.

References

[1] A. Aiken and A. Nicolau, "A Development Environment for Horizontal Microcode," *IEEE Transactions on Software Engineering*, May 1988, pp. 584-594.

[2] M. Butler, T.-Y. Yeh, Y.N. Patt, M. Alsup, H. Scales, and M.C. Shebanow, "Single Instruction Stream Parallelism Is Greater Than Two," *Proc. ISCA*, Toronto, 1991, pp. 276-286.

[3] K. Ebcioglu, "Some Design Ideas for a VLIW Architecture for Sequential Natured Software," in M. Cosnard, *et al.*, (eds.), *Parallel Processing (Proc. IFIP WG 10.3 Working Conference on Parallel Processing, Pisa, Italy)*, North Holland, 1988, pp. 3-21.

[4] M. Franklin and M. Smotherman, "A Fill-Unit Approach to Multiple Instruction Issue," *Proc. Micro-27*, San Jose, November 1994, pp. 162-171.

[5] A. Glew, personal communication.

[6] W.-M. Hwu and Y.N. Patt, "HPSm, A High Performance Restricted Data Flow Architecture Having Minimal Functionality," *Proc. ISCA*, Tokyo, 1986, pp. 297-306.

[7] S.W. Melvin, M.C. Shebanow, and Y.N. Patt, "Hardware Support for Large Atomic Units in Dynamically Scheduled Machines," *Proc. Micro-21*, San Diego, December 1988, pp. 60-66.

[8] Y.N. Patt, W.-M. Hwu, and M.C. Shebanow, "HPS, A New Microarchitecture: Rationale and Introduction," *Proc. Micro-18*, Asilomar, December 1985, pp. 103-108.

[9] Y.N. Patt, S.W. Melvin, W.-M. Hwu, and M.C. Shebanow, "Critical Issues Regarding HPS, A High Performance Microarchitecture," *Proc. Micro-18*, Asilomar, December 1985, pp. 109-116.

[10] Y.N. Patt, S.W. Melvin, W.-M. Hwu, M.C. Shebanow, C. Chen, and J. We, "Run-Time Generation of HPS Microinstructions From a VAX Instruction Stream," *Proc. Micro-19*, New York, October 1986, pp. 75-81.

[11] J. Pierce and T. Mudge, *IDtrace - A Tracing Tool for i486 Simulation*, Technical Report CSE-TR-203-94, Dept. Of Electrical Engineering and Computer Science, University of Michigan, 1994.

[12] J.E. Wilson, S.W. Melvin, M.C. Shebanow, W.-M. Hwu, and Y.N. Patt, "On Tuning the Microarchitecture of an HPS Implementation of the VAX," *Proc. Micro-20*, Colorado Springs, December 1987, pp. 162-167.

[13] P6 description available at http://www.intel.com.

Appendix: Example instructions mapped onto microoperations

These are example mappings that do not represent the P6 or any particular x86 implementation.

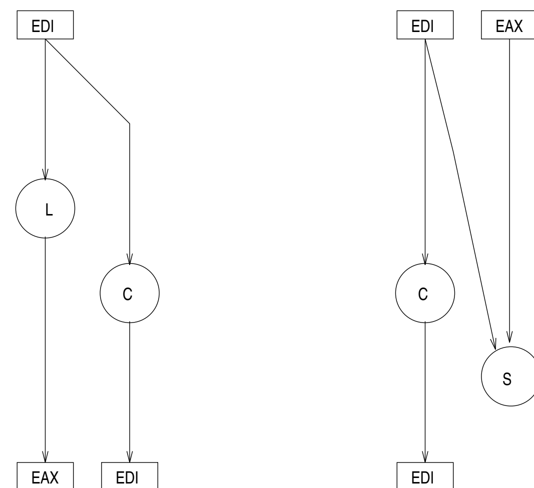

Figure 9: LODS (left) and STOS (right). We do not show the increment/decrement dependence on the direction flag.

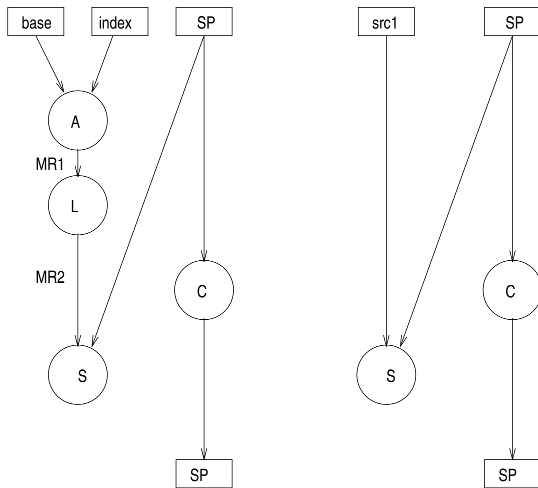

Figure 10: PUSH mem (left) and PUSH reg (right).

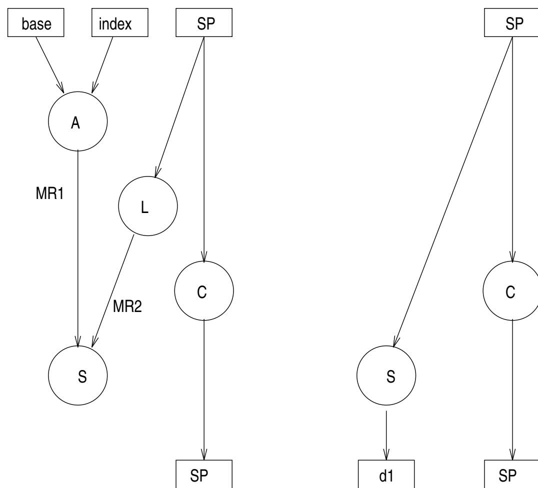

Figure 11: POP mem (left) and POP reg (right).

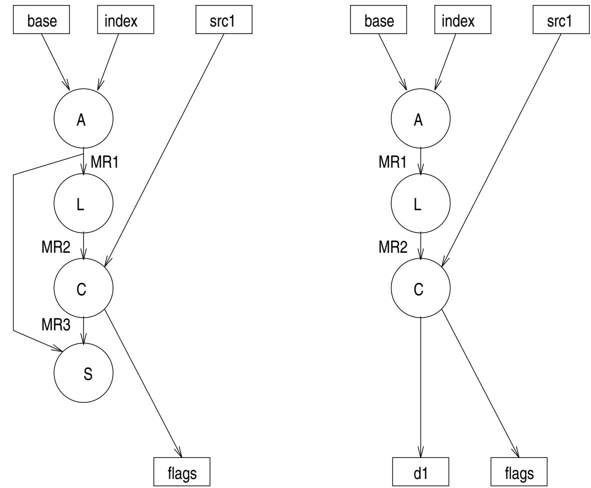

Figure 12: ADD mem,reg (left) and ADD reg,mem (right). ADC has two similar forms with the flags register used as another input. Multiply and divide are also similar with multiply producing two destination registers and setting the flags register and divide needing a second source register, for double-length product and double-length dividend, respectively.

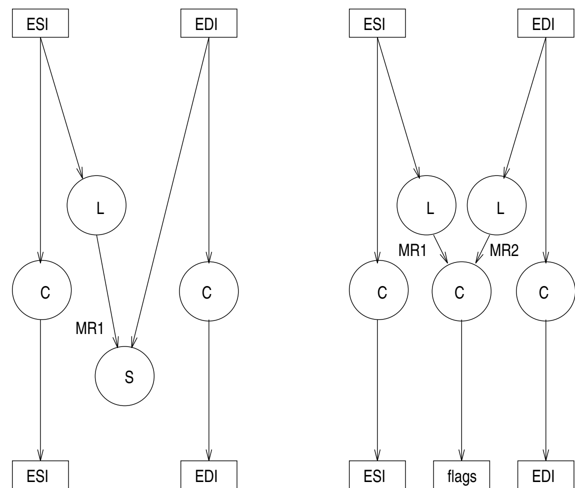

Figure 13: MOVS (left) and CMPS (right). We do not show the increment/decrement dependence on the direction flag.