Reducing the Delay for Decoding Instructions by Predicting Their Source Register Operands

Sanghyun Park, Jaeyung Jun, Changhyun Kim, Gyeong Il Min and Hun Jae Lee and Seon Wook Kim *

School of Electrical and Computer Engineering, Korea University, Seoul 02841, Korea; dasa00@korea.ac.kr (S.P.); coole923@korea.ac.kr (J.J.); hunterkch@korea.ac.kr (C.K.); gyeong9m@korea.ac.kr (G.I.M.); hjlee3031@korea.ac.kr (H.J.L.)
* Correspondence: seon@korea.ac.kr; Tel.: +82-2-3290-3251

Received: 13 April 2020; Accepted: 14 May 2020; Published: 16 May 2020

Abstract: The fetched instructions would have data dependency with in-flight ones in the pipeline execution of a processor, so the dependency prevents the processor from executing the incoming instructions for guaranteeing the program’s correctness. The register and memory dependencies are detected in the decode and memory stages, respectively. In a small embedded processor that supports as many ISAs as possible to reduce code size, the instruction decoding to identify register usage with the dependence check generally results in long delay and sometimes a critical path in its implementation. For reducing the delay, this paper proposes two methods—One method assumes the widely used source register operand bit-fields without fully decoding the instructions. However, this assumption would cause additional stalls due to the incorrect prediction; thus, it would degrade the performance. To solve this problem, as the other method, we adopt a table-based way to store the dependence history and later use this information for more precisely predicting the dependency. We applied our methods to the commercial EISC embedded processor with the Samsung 65nm process; thus, we reduced the critical path delay and increased its maximum operating frequency by 12.5% and achieved an average 11.4% speed-up in the execution time of the EEMBC applications. We also improved the static, dynamic power consumption, and EDP by 7.2%, 8.5%, and 13.6%, respectively, despite the implementation area overhead of 2.5%.

Keywords: register operand detection; data dependency; decoding; ISAs

1. Introduction

Most processors [1–5] adopt a pipeline architecture to increase their operating frequency by dividing the execution flow into multiple stages and improve execution throughput by executing them independently. However, during the pipeline execution, control and data dependency between newly fetched and in-flight instructions may occur, which makes an instruction scheduler insert stalls into the pipeline for guaranteeing the program correctness, and therefore degrades the execution throughput. To minimize the control hazard, a branch predictor [6] has been widely adapted to predict a target address of a branch instruction from the previous branch history, and the processor fetches instructions from the predicted target address. If the branch predictor mispredicts, the processor flushes the speculatively executed instructions and starts to fetch new instructions from the correct target address. The prediction accuracy is critical to the performance due to the flush overhead. On the other hand, to resolve the data dependency, the processor identifies register names or memory addresses of the source and destination operands from the in-flight instructions. An in-order processor inserts stalls to the pipeline if the data dependency is detected, and an out-of-order processor would issue independent instructions instead of pipeline stalls for delivering higher performance.
Each instruction represents the source and destination operands at its specific bit-field positions, so the processor can detect the register dependence after decoding the fetched instruction thoroughly and comparing it with in-flight instructions in the pipeline. The decoder delay to support complicated instructions can be lengthy, so the dependence detection logic would become recognized as a critical path. Especially in a small embedded processor that supports as many ISAs as possible to reduce code size, the delay increases, and eventually, the operating frequency of the processor decreases. In reality, in the case of the commercially available EISC processor [7], from our experiment, we found that the register dependence detection unit was at the critical path in the design. More precisely, its delay accounted for about 33.8% of the critical path delay, approximately 49.4% of which was consumed in the logic to identify the source register operands. Also, we found that we could increase the operating frequency by up to 16.6% if we remove the dependence logic from the critical path.

In this paper, we alleviate the delay by predicting ISA source register operands. In the EISC ISA, which is our research target, we observed that most instructions describe their source register operands at the following four bit-field positions: [8:5], [7:4], [3:1], and [3:0]. Based on the observation, we propose an ISA pre-decoding scheme that each instruction is supposed to represent its source register operands at all the above four positions rather than finding ones precisely by the complete instruction decoding. This scheme identifies the register dependence by comparing the assumed register values with the destination register operands of the in-flight instructions. If the instruction does not include any source operand, we consider that it does not involve any data dependency. Also, some EISC instructions use an encoded bit vector for representing multiple source register operands at once, and such instructions are always assumed to have the data dependency on in-flight instructions to eliminate the delay of decoding the bit vector. However, this case is infrequent so that the performance would not be affected. What we need to pay attention to is that speculating more source register operands than necessary incurs unnecessary stalls, so degrades the performance. Even worse, due to the temporal locality of the executed instructions, the stalls would repeatedly occur. Therefore, we propose a table-based scheme to store the dependence history and later use this information for more precisely predicting the dependence; thus, we would eliminate the unnecessary data hazard from incorrect speculation.

When we applied our two schemes to the EISC embedded processor and synthesized it with the Samsung 65 nm process, we found that its area was increased by about 2.5%, but its maximum operating frequency was improved by approximately 12.5%; Also, from the execution of the EEMBC benchmark [8], the frequency improvement made us achieve 11.4% speedup on the average with only 0.2% increment of the stall cycles. The reduced critical path delay resulted in the reduction of static power and dynamic power consumption, and EDP by 7.2%, 8.5%, and 13.6%, respectively. Our methods needed the implementation cost. However, because the path reduction made the synthesis tool use more efficient cells, we could reduce power consumption and energy.

The main contributions of this paper are as follows.

- To out best of the knowledge, our study is the first to predict source and destination operands in ISAs in a decode stage to reduce the decoder delay with minimal overhead, instead of adding pre-decoding logic in a fetch stage like in the previous works.
- For this purpose, we propose the pre-decoder to predict the source and destination operands’ positions in ISAs and the table-based data hazard predictor to remove the incorrect pre-decoding, thus minimizing the unnecessary data hazard.
- We applied our schemes to the commercially available EISC embedded processor and showed the applicability of our study to real products.

This paper consists of the following: Section 2 introduces prior studies related to this work. Section 3 describes the motivation of our work, and Section 4 presents our two schemes with their implementation. Section 5 evaluates their performance regarding speedup, power, implementation cost, and so on. Finally, the conclusion of this paper is given in Section 6.
2. Related Work

In general, a method to increase the operating frequency of a processor is to add more pipeline stages [9]. However, this method would consume more power and make the implementations for removing or hiding stalls caused by dependence complicated. A typical technique of hiding latency from the data dependency is a register forwarding [10], where its propagation logic passes execution or memory access results directly to the functional units that need them.

There are several studies about the pre-decoding for reducing the decoding latency, and most of them placed a pre-decoding unit in the fetch stage instead of the decode stage for supporting the decoder execution, not for actual decoding. The authors of Reference [11] places a pre-decoding unit between the main memory and an instruction cache and identifies the instruction type. The pre-decoded type information allows the instructions to be quickly delivered to the associated functional units, resulting in the improved operating frequency of a processor. MIPS R1000 pre-decoded instructions and rearranged source and destination fields to be in the same position for every instruction for the easy decoding in the decode stage [12]. Similarly, x86 processors have used the pre-decoding, which assesses the length of instruction so the subsequent decoders can handle it efficiently since x86 instructions can vary from one to multiple bytes in length [13]. The branch predictor needs to determine whether an instruction is a branch, and if so, its branch type for selecting the corresponding predicted target. This decision makes the decode logic lie in the critical path, so the predictor pre-decodes the instructions before storing instructions in an instruction cache [14]. All of the approaches needed an additional decoder for the pre-decoding in the fetch stage. Also, they needed extra memory space for storing special bits per instruction for the decoded information and aligned instructions. Therefore, they resulted in a high area and power overhead. Santana et al. [15] stored the decoded instructions in the memory hierarchy instead of using the extra buffer.

Another use of the pre-decoding is for avoiding the repeated decoding the same instruction. The authors of Reference [16] proposed a hardware folding technique that dynamically transforms Java bytecodes groups into RISC instructions, storing them in a cache to enable reuse.

In our study, we get a hint from the instruction structure and predict the data dependency between instructions, so reducing the latency of the decoder without a large amount of additional circuitry or memory.

3. Motivation

3.1. EISC Processor Architecture

Figure 1 shows the commercially available simple embedded EISC processor architecture used in this study [7]. The processor supports a single-issue in-order execution with 16-bit instructions and 32-bit data width and consists of four-stage pipelines: FE (Fetch), DEC (Decode), EX/WB (Execution/Write-Back), and MEM/WB (Memory/Write-Back).

The FE stage predicts a target address in an “always not taken” manner for fetching instructions from the instruction memory without using an advanced branch predictor to minimize the implementation cost. If a branch miss-prediction occurs, the pipeline controller flushes the speculatively executed instructions and causes the FE stage to fetch the instruction from the correct target address. The DEC stage decodes the fetched instruction, and its register dependence unit checks whether there is a RAW register dependence between the decoded instruction and the instructions being executed in the EX and MEM stages. If there is no dependence, the pipeline controller issues the decoded instruction to the EX stage; Otherwise, the controller inserts a stall into the EX stage to ensure execution correctness. The processor does not support the data forwarding logic due to its design complexity. The EX stage supports 32-bit integer ALUs consisting of multi-cycle multiplier and divider. Finally, the MEM stage contains logic for accessing data memory through load and store commands. The instruction can be retired at either EX or MEM stages.
One of the most critical factors in determining the implementation cost of a small embedded system would be a memory; so, an embedded processor, in general, supports complex instructions to use small code size. Therefore, the decode stage becomes more complicated and lengthy in time, and its logic is identified as a critical path in our target EISC core [17] as an example.

![Diagram of EISC processor architecture]

**Figure 1.** The embedded EISC processor architecture. The dark shaded components, “instruction decoder” and “history table” were modified and added for our study, respectively.

### 3.2. EISC Instruction Set Architecture

We analyzed the EISC ISAs into the following three categories by the characteristics of the bit field, which refers to a source register operand: Source Field (SF), Source-Less (SL), and Encoded Field (EF). Table 1 shows the classified results, and the number of bit fields and the corresponding number of instructions.

| Category           | Bit-Field Positions | # of Instructions |
|--------------------|---------------------|--------------------|
| Source Field (SF)  | [8:5]               | 1                  |
|                    | [7:4]               | 4                  |
|                    | [7:4], [3:0]        | 21                 |
|                    | [7:4], [[3:1],0] [[3:1],1] | 2             |
|                    | [3:0]               | 33                 |
| Source-Less (SL)   | None                | 37                 |
| Encoded Field (EF) | [7:0]               | 4                  |
| Total              | -                   | 102                |

The Source Field (SF) contains instructions to identify one and more source register operands from their particular bit-field positions, [8:5], [7:4], [3:1], and [3:0]. In our proposal, we do not fully interpret them at the decode stage but assume that all the specific bit-fields represent the source register operands. Then, we compare the assumed source register values with the destination ones of the in-flight instructions for detecting the data dependency at the pipeline execution. More than half of the total ISAs belong to this category. The Source-Less (SL) is a category containing instructions which do not have any source register operands, which typically includes a direct branch and the LERI instructions [18]. Therefore, the SL instructions do not involve any data dependency, so dependence detection is not required. The Encoded Field (EF) includes instructions where multiple source register operands
are encoded and described as a bit vector. The EF contains multiple memory access instructions by auto-increment/decrement operations, which can use up to eight source register operands. That is, the bit vector must be decoded to know the source registers. However, in our approach, we assume that such instructions always have the data dependency on in-flight instructions for eliminating the delay of decoding the bit vector.

Figure 2 shows the dynamic instructions regarding the three categories when we ran the EEMBC benchmarks [8].

Most dynamic instructions belong to the SF category, so the processor needs to compare the values from the four predetermined bit-field positions with the destination register operands. The SL instructions occupied about 25.4% of the total ISA execution. Fortunately, the processor does not concern about any of their data dependency if it knows them belonging to the SL at their decoding. For the EF instructions, we assumed that they always have a data dependency on in-flight instructions to avoid complex decoding. However, since their occurrence is infrequent, about 1.1%, the performance degradation due to the assumption would be insignificant.

In order to support our motivation further, we also analyzed the core instruction set of the MIPS 32-bit integer ISA [19] and the RISC-V 32-bit integer base ISA [20], and Table 2 shows our analysis results. In MIPS and RISC-V ISAs as well as EISC, most instructions acquire source register operands in specific bit-fields of the instruction (SF). For MIPS, these bit-fields are [25:21], [20:16], and for RISC-V, they are [27:24], [24:20], and [23:20], [19:15]. The proportion of instructions that do not have a source register operand (SL) is 6.5% and 13.6% for MIPS and RISC-V, respectively. Finally, the instructions which require the decoding of specific bit-fields to find the source register operands do not exist for MIPS, and RISC-V accounts for 13.6% for the total ISA. Therefore, most instructions in MIPS and RISC-V also belong to the SF, and since the number of their bit-fields is less than four, MIPS and RISC-V, as well as EISC, can fully support our motivation.

| Processors | MIPS | RISC-V |
|------------|------|--------|
| Category   | Bit-Field Positions | # of Instructions | Bit-Field Positions | # of Instructions |
| SF         | [25:21], [20:16]    | 12                | [24:20], [19:15]    | 19                |
|            | [25:21]             | 17                | [19:15]             | 12                |

Table 2. Category of the MIPS and RISC-V instructions depending on their source register operand bit-fields.
3.3. Opportunity to Reduce the Incorrect Pre-Decoding

Generally, in embedded applications, instructions are executed repeatedly, and as a result, if a data dependency associated with the instructions are mistakenly guessed, unnecessary stalls frequently occur. Therefore, to eliminate such occurrences, we considered a method of storing the data dependence history in a table and using the history information later. We traced the Program Counter (PC) of the instruction by executing the EEMBC program and measured the hit ratio of the table accesses with different table sizes. We assumed that the table was fully-associative.

Figure 3 shows that the hit ratio of the 128 entry-table was about 67.2% on the average, which was close to the unlimit with a difference of about 13.6%. We decided that the size was the most appropriate for our design. In the applications, such as canrdr01, rspeed01, tblook01, a2time01, and puwmod01, their hit ratios were low, since 32.2%, 79.8%, 12.1%, 37.5%, and 100.0% of the total dynamic instructions were executed only once, respectively, as shown in Figure 4.

![Figure 3](image1.png)

**Figure 3.** Hit ratio with respect to various table sizes.

![Figure 4](image2.png)

**Figure 4.** Ratio of only once executed dynamic instructions of the EEMBC applications.
4. Reducing the Delay in a Decode Stage

4.1. Pre-Decoding Scheme

The existing decoder decodes instructions completely at the decode stage; thus, it identifies their source register operands and compares them with the destination register operands of the in-flight instructions for detecting their data dependency. Figure 5a shows a code example to involve data dependency: $I_3$ is decoded fully at the decode stage. After that, its source register operand, $r_1$, is identified and compared with the destination register operands of $I_2$ and $I_1$ (in-flight instructions), $r_1$ and $r_4$, respectively, in sequence. The decoder matches $r_1$ between $I_2$ and $I_3$, so the pipeline inserts stalls due to their data dependency.

Our pre-decoding scheme does not fully decode instructions to identify the source register operands; instead, it obtains them from all the predetermined bit-field positions, that is, [7:4], [8:5], [3:1], and [3:0] of instructions as shown in Table 1. Then, the pipeline compares all of the pre-decoded source register operands with the destination register operands of the in-flight instructions to detect their data dependency, and it is shown in Figure 5b. When decoding $I_3$ (0xf023), we assume the source register operands from all the predetermined bit-fields, thus obtains four possible source operands (2, 1, 3, 3). Then, the pipeline compares them with the destination register operands of $I_2$ and $I_1$ (in-flight instructions), $r_1$ and $r_4$, respectively, in sequence.

![Figure 5](image-url) (a) An example of the code and (b) the data dependence detection for the SF instructions in the proposed scheme.

Since the number of instructions belonging to the SF category is large, the source register operands are recognized in the predetermined bit-fields at the decoding stage. The SL instructions are interpreted as soon as possible, that is, at the fetch stage. Since the number of EF instructions is minimal, it is not difficult to recognize the type of instructions at the decoding stage. Our scheme does not affect the program’s correctness because the source register operand is always included in all the four
predetermined bit-field values. Also, since we do not apply full instruction decoding, we can reduce its associated circuit delay. In a typical processor, the number of bit-fields used in the source register operand would not be significant because the decoding logic must be simple to implement. However, our method interprets more source register operands than what one instruction represents; thus, it can incur unnecessary stalls and consequently would increase the total execution cycle.

The instructions belonging to the SL category do not have any source register operand. Therefore, if the pipeline knows that an instruction running at the decode stage belongs to the SL, the pipeline can issue the instruction without detecting a data dependency. To apply this to the EISC processor, we made the fetch stage identify whether or not the instruction belongs to the SL, and passed the identified information to the decode stage. The added SL identification unit does not affect the critical path. The EF instruction has an encoded bit vector to describe several source register operands at one time. In this case, particular bit-fields cannot be regarded as a source register operand, and the bit vector must be decoded to find the source register operands, which loses our delay gain in the simplified SF and SL execution. Therefore, to maximize the gain, we suppose that the EF instructions always have a dependence on the in-flight instructions. However, since the EF execution is rare, the performance degradation due to this very conservative assumption can be ignored, as discussed in Section 3. Since the number of EF instructions is small and we can find out whether an instruction belongs to the EF category with almost no implementation overhead, we implemented the EF related logic to the decode stage.

Figure 6 shows the hardware logic about the pre-decoding scheme. For the SF instructions, the existing hardware was modified to compare all four bit-fields for possible source register operands with the destination register operands of the in-flight instructions at the same time. If at least one pair is matched, a data hazard is generated to the pipeline.

Figure 6. Implementation of the instruction pre-decoding logic in the EISC processor.

4.2. Table-Based Scheme

Figure 4 showed that the instructions were executed repeatedly, which would result in many unnecessarily repeated stalls due to the incorrect assumption by the pre-decoding. To overcome the problem, we propose a table-based scheme to store the incorrect prediction and use the information
for the later execution. We do not record the history of the dependence of every instruction because it requires a large history table. Because our pre-decoding accuracy is high, we manage only the wrong dependence history and can achieve significant performance benefits even with a small table.

We made the history table as a fully-associate cache, but its complexity is minimal, as shown in Figure 7. The table consists of 4-entries, one of which has an 11-bit tag, 1-bit valid, and 32-bit pattern history. Therefore, each entry can record the unnecessary data hazard history of 32 instructions, so our table can do up to a total of 128 instructions. We marked the occurrence of unnecessary data hazards as 1b’1 per instruction. Otherwise, we did as 1b’0.

Figure 7. The prediction table used in the data hazard predictor implemented in the EISC processor.

Because the program memory size of the EISC processor was 128 KB and the instruction size was 16-bit, the processor used 17-bit of PC, and its LSB was always 0. Also, since the size of the pattern history of each entry was 32-bit, 5-bit was used as an index for searching the entry. Therefore, the top 11-bit of the PC was used as a tag. Also, 44-bit was used for each entry by combining the tag, the valid, and the pattern history. As a result, we implemented the history table with a total of 176-bits.

Algorithm 1 shows the insert and lookup functions for the history table. Whenever the pre-decoding scheme has issued the unnecessary data hazard, it sets the corresponding bit field in the history table by calling insertHistoryTable. If the access to the table is miss, the new entry is overwritten to the LRU entry (Lines 5∼6). For example, in Figure 7, the pre-decoding scheme predicted at the previous execution incorrectly and assumed that there was a RAW dependency between instructions from PC_i + 2 to PC_i + 4. Thus, to indicate the incorrect prediction, the function, insertHistoryTable is called, and the corresponding bit field is set. There is a RAW dependency between instructions from PC_i + 4 to PC_i + 6. However, in this case, the pre-decoding scheme predicted the dependency correctly at the previous execution. Thus, the history bit needs to be set to 0 even though there is a RAW dependency. So, the function is not called, since the default value is 0.

During the history table lookup, if the target PC hits (Lines 9∼10) and the returned value is 1, we know that the pre-decoding scheme has issued the unnecessary data hazard before due to miss-prediction. If the returned value 0, it implies either no history or correct prediction from the previous execution. Therefore, we need to ignore the current incorrect dependence prediction for avoiding performance degradation. If the history table access is miss, the prediction result from the pre-decoding scheme is used (Line 12). When a branch instruction is executing at the execution stage, we do not record its related dependence to guarantee the program’s correctness.
Algorithm 1: Insert and lookup functions for the prediction table.

// Initially all the valid bits are 0's.

1 void insertHistoryTable(PC) // called when unnecessary dep is identified
2 if PC[16 : 6] is hit then
3     // one of the tags in the table is matched and its valid bit is 1
4         set a bit field according to PC[5 : 1] of the matched pattern history;
5 else
6     // miss: make a new history entry
7         store PC[16 : 6] into a tag of the LRU entry, and set its valid bit;
8         set a bit field according to PC[5 : 1] and unset all the other bit fields in its pattern history;
9 end

8 boolean lookupHistoryTable(PC)
9 if PC[16 : 6] is hit then
10     // one of the tags in the table is matched and its valid bit is 1
11         // 0: necessary data hazard or no history (Use the pre-decoding scheme)
12         // 1: unnecessary data hazard issued before
13         return a bit field according to PC[5 : 1] of the matched pattern history;
14 else
15     // miss: history is not found. Use the pre-decoding scheme.
16     return 0;
17 end

5. Performance Analysis

5.1. Experimental Methodology

We implemented and verified the single issue, and in-order 4-stage pipelined EISC processor with the proposed schemes onto the Xilinx Kintex-7 evaluation platform [21]. Since the EISC processor targets the embedded system, we measured its performance using the EEMBC benchmark suite [8] while excluding applications whose code and data sizes exceeded our memory sizes shown in Figure 1. We also implemented hardware performance counters (HPCs) inside the EISC processor to measure the execution and stall cycles of the applications in detail. In addition to the functional verification, we used the Samsung 65 nm process standard cell [22] and the Design compiler [23] for measuring its maximum operating frequency of the processor and implementation cost.

5.2. Performance Evaluation

We measured the execution and stall cycles by using the following three configurations: (1) the baseline EISC processor, (2) by adopting the pre-decoding scheme to the baseline one, and (3) by taking both the pre-decoding and the history schemes to the baseline one. Figure 8 shows the total execution cycles of our approaches, which are normalized by the baseline configuration. In our approaches, the total cycles increases due to stalls from the operand misprediction, and Figure 9 shows the increment ratio of the stall cycles in each instruction category. It should be noted that in our approaches, the total cycles increases, but the execution time decreases by increasing an operational frequency, and it is described in Section 5.3.
Figure 8. The total cycle overhead of our approaches when running the EEMBC applications. All the measurements were normalized with that of the baseline EISC processor.

Figure 9. Ratio of stall cycle increment to the baseline EISC processor.

The average execution cycle by the pre-decoding scheme was increased by about 2.7% compared to the baseline EISC processor due to that that we incorrectly predicted the source register fields and we made the assumption of dependence in all the EF instructions. As a result, we could not issue independent instructions of the in-flight instructions, and they incurred unnecessary data hazards in the pipeline. However, the pre-decoding with the history table significantly reduced the overhead, and only increased the execution cycles by about 1.0% on average. Also, we found that the increment ratio of the stall cycles was 1.6% by the pre-decoding scheme and 0.2% by the pre-decoding with the history table.

Especially in autocor00, routelookup, fft00, idctrn01, and bezier01fixed, the increment of the execution cycles by only the pre-decoding scheme was significantly decreased. The applications included many loop iterations and incurred many unnecessary data hazards, so there were many opportunities for applying the history table to mitigate the increment of the stall cycles. The applications showed 98.6% hit ratio to the table. However, in the case of canrdr01, rspeed01, tblook01, a2time01, and puvmod01, they had poor temporal locality of instructions, and their hit ratio was only 11.2%. Therefore, the predictor could not mitigate the overhead of the increased stall cycles. In fbital01, convent00, bitmnp01, dither01, rotate01, and ospf, although they exploited high temporal locality, there were few additional data hazards; thus, the opportunity for the history table was also insufficient.

From Figure 9, we found that, for the SF instructions, the pre-decoding scheme caused unnecessary data hazards in the overall EEMBC applications, which was particularly noticeable in autocor00, routelookup, fft00, idctrn01, and bezier01fixed. The overhead of the SF instructions due to the pre-decoding scheme was the largest in idctrn01, and the ratio was about 6.6%. However, the applications exploited high temporal locality; thus, the related overhead was decreased remarkably after applying the history table. We pre-decoded the SL instructions and did not cause any data hazard in the pipeline;
thus, there was no increment due to the SL instruction. However, in the case of the EF instructions, we always issued a data hazard; therefore, their execution overhead was significant, noticeably in such applications with a large proportion of EF instruction as fft00, idctrn01, bezier01fixed, tblook01, and a2time01. However, autcor00 showed no increment by the EF instructions despite the large proportion of the EF instructions in the entire dynamic instructions. This result was why most EF instructions already had a data dependency with in-flight instructions. From the history table, the gain was low, that is, about 0.1%. In the SF instructions, autcor00, routelookup, fft00, idctrn01 and bezier01fixed applications exploited high temporal locality; thus, the related overhead was decreased remarkably after applying the history table.

5.3. Operational Frequency and Speedup

In the baseline EISC processor, we identified the path that performs the data dependency detection after instruction decoding as the critical path. Therefore, by pre-decoding source register fields, we could remove the detection logic from the critical path of the EISC processor, which increased the maximum operating frequency by 12.5%, that is, 370 MHz to 416.6 MHz.

Figure 10 shows the speedup of the EEMBC applications by improving the frequency. Since the history table was not in the critical path, only one bar was presented. We could achieve an average speed of about 11.4%, and especially in autcor00, fbital00, conven00, and bitmnp01, we achieved speedup as 12.5% maximum since we almost removed the unnecessary stall cycles.

5.4. Area and Power

To analyze the hardware overhead of the proposed EISC processor, we synthesized the baseline and our EISC processors with the same timing constraint for evaluating their area, static and dynamic power consumption.

We incurred insignificant area overhead, that is, by 2.5% due to the pre-decoding and the history table logic. However, our proposed schemes had an advantage of the power consumption over the baseline processor. Figure 11 shows the static power per unit area of the NAND gate with respect to its driving strength, which shows that the NAND gate tends to consume more static power per unit area when the driving the gate becomes stronger. It means that when the logic area of a design with a strong driving strength gate and a design with a weak driving strength gate are the same, the latter consumes less static power than the former. In the case of the EISC processor, we had a better timing margin than the baseline EISC processor since we improved its critical path delay. Therefore, the EDA tool used a weak driving strength gate to synthesize our EISC processor. Therefore, the static consumption of the proposed approach was lower than that of the baseline EISC processor by about 7.2%.

Our EISC processor searched for the unnecessary data hazards by using the history table in the fetch stage. Therefore, the dynamic power consumption of the fetch stage was increased by 8.2% due
to the table implementation. However, the dynamic power consumption of the decoding stage was decreased by 16.7% by simplifying the decoding stage through the pre-decoding scheme and using the weak driving strength gates. Consequently, the total dynamic power consumption was reduced, that is, by 8.5%.

![Figure 11](image1.png)

**Figure 11.** Normalized static power per unit area in 2-input NAND gate. All the results were normalized to the X1.

### 5.5. Energy and EDP

The operational frequency improved by 12.5%, so it reduced the execution time. Also, the synthesis tool used the weak driving strength gates, thus reducing power consumption. Therefore, our approach could reduce energy consumption and improve EDP. Figure 12 shows that the energy consumption of the EEMBC applications by our proposed EISC processor is normalized to the baseline result. Our EISC processor energy consumption was improved by about 4% on average compared to the baseline. The figure shows that the higher the speedup ratio of the EEMBC applications, the higher the energy improvement ratio.

![Figure 12](image2.png)

**Figure 12.** Energy consumption of the EEMBC applications with respect to the baseline EISC processor.

Figure 13 shows the normalized EDP of our EISC processor with respect to the baseline one, and the EDP of our processor was improved by about 13.6% on average. When considering Figure 10, we found that the more significant the speedup of the EEMBC application, the higher the EDP improvement ratio. The highest EDP improvement ratio, 15.3% was achieved in `autcor00`, `fitbal00`, `convenc00`, `bitmnp01`, which had the highest speedup gain. On the other hand, the lowest EDP improvement ratio was 10.3% in `bezier01fixed`, which had the smallest speedup gain.

The reciprocal of EDP would imply the metric, performance per power. Since our EDP is less than that of the original design in all cases, our design would be fit for low-powered IoT and embedded systems.
6. Conclusions

Data dependency can occur between in-flight instructions in the pipeline stages of the processor. It is, therefore, necessary to detect the dependence between them by identifying their source and destination operands. The existing decoding method, which requires the complete decoding, increases the decoding latency when supporting complex ISAs; it would affect the critical path of the pipeline.

In order to solve this problem, we proposed “a pre-decoding” method that performs data dependency detection by assuming the source register operand bit-fields based on the analysis result of ISAs without completely decoding the instructions. We analyzed the EISC ISAs and the dynamic instructions of the EEMBC applications for justifying our motivation. However, the pre-decoding method caused unnecessary stalls due to the incorrect assumption, which degraded the performance. To solve this problem, we adopted a table-based way to store the dependence history and later used this information for more precisely predicting the dependence.

We modified the data dependence logic and the pipeline stages of the EISC processor to apply for our schemes. We measured the performance of the EISC processor on the FPGA platform, and we synthesized the EISC processor using the standard cell of the 65nm process to measure the maximum operating frequency of the processor and analyze the hardware overhead of the proposed scheme. As a result of applying the proposed methods to the EISC processor, we improved the critical path delay of the processor by about 12.5%, so achieved an average of 11.4% speedup with a small hardware area overhead of 2.5%. The static, dynamic power consumption and EDP of the processor were improved by 7.2%, 8.5%, and 13.6%, respectively.

Our study was the first to predict and interpret source and destination operands in ISAs for reducing the decoding delay and minimizing the related overhead. As we discussed in the motivation section, other processors’ ISAs have similar structural features to the EISC ISA. Therefore, we would apply our technology to other processors and make it useful. Also, the pattern history can work well with 32 statically consecutive instructions. Otherwise, the miss ratio in the history table increases, so incurring more stalls. In order to solve the problem, the structure of hardware would be modified so that the hardware cost would be increased. For example, we would increase the table size and the set associativity. Instead, we have a plan to use a software method, that is, tracing the executed instructions at runtime and using the profiled result for a compiler’s jump optimization. The plan is left as our future work.

Author Contributions: S.P. mainly performed the research for this paper, J.J., C.K., G.I.M., H.J.L. and S.W.K. supported his research in different ways. Also, S.W.K. leds this project, and therefore he is responsible for this research and publication. All authors have read and agreed to the published version of the manuscript.

Funding: Ministry of Trade, Industry and Energy: 10052716 and N0001883.
Acknowledgments: This work was supported by the IT R&D program of MOTIE/KEIT [10052716, Design technology development of ultra-low voltage operating circuit and IP for smart sensor SoC] and the Competency Development Program for Industry Specialists of the Korean Ministry of Trade, Industry and Energy (MOTIE), operated by Korea Institute for Advancement of Technology (KIAT) (No. N0001883, HRD Program for Intelligent Semiconductor Industry). The chip fabrication and EDA tool for this work were supported by the IC Design Education Center (IDEC), Korea.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. ARM. A9 Processor. Available online: https://developer.arm.com/products/processors/cortex-a/cortex-a9 (accessed on 19 December 2018).
2. Samsung. Samsung Exynos 9810. Available online: https://www.samsung.com/semiconductor/minisite/exynos/ (accessed on 19 December 2018).
3. Qualcomm. Qualcomm Snapdragon 845. Available online: https://www.qualcomm.com/snapdragon (accessed on 19 December 2018).
4. AMD. AMD Takes Computing to a New Horizon with Ryzen Processors. Available online: https://www.amd.com/en/press-releases/amd-takes-computing-2016dec13 (accessed on 19 December 2018).
5. Hammarlund, P.; Kumar, R.; Osborne, R.B.; Rajwar, R.; Singhal, R.; D’Sa, R.; Chappell, R.; Kaushik, S. Haswell: The Fourth-Generation Intel Core Processor. IEEE Micro 2014, 34, 6–12. [CrossRef]
6. Lee, C.C.; Chen, I.C.K.; Mudge, T.N. The Bi-mode Branch Predictor. In Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO ’97), Research Triangle Park, NC, USA, 3 December 1997; pp. 4–13.
7. ADChips. EISC Core. Available online: http://adc.co.kr/ (accessed on 19 December 2018).
8. EEMBC. EEMBC Benchmark Suite. Available online: https://www.eembc.org/products/#single (accessed on 19 December 2018).
9. Sprangle, E.; Carmean, D. Increasing Processor Performance by Implementing Deeper Pipelines. In Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA ’02), Anchorage, AK, USA, 25–29 May 2002; pp. 25–34.
10. Hennessy, J.; Patterson, D. Computer Architecture: A Quantitative Approach; Morgan-Kaufman: Waltham, MA, USA, 2011.
11. DeLano, E.R.; Gleason, C.A.; Forsyth, M.A. Predecoding Instructions for Supercalar Dependency Indicating Simultaneous Execution for Increased Operating Frequency. U.S. Patent 5,337,415, 4 December 1992
12. Yeager, K. The MIPS R10000 Superscalar Microprocessor. IEEE Micro 1996, 16, 28–41. [CrossRef]
13. Singh, T.; Rangarajan, S.; John, D.; Schreiber, R.; Oliver, S.; Seahra, R.; Schaefer, A. Zen 2: The AMD 7nm Energy-Efficient High-Performance x86-64 Microprocessor Core. In Proceedings of the 2020 IEEE International Solid-State Circuits Conference (ISSCC ’20), San Francisco, CA, USA, 16–20 February 2020.
14. Wu, D.; Aasaraai, K.; Moshovos, A. Low-cost, high-performance branch predictors for soft processors. In Proceedings of the 23 rd International Conference on Field programmable Logic and Application (FPL ’13), Porto, Portugal, 2–4 September 2013.
15. Santana, O.J.; Falcón, A.; Ramírez, A.; Valero, M. Branch predictor guided instruction decoding. In Proceedings of the 15th international conference on Parallel architectures and compilation Techniques (PACT ’06), Seattle, WA, USA, 16–20 September 2006; pp. 202–211.
16. Sideris, I.; Pekmestzi, K.; Economakos, G. A Predecoding Technique for ILP Exploitation in Java Processors. J. Syst. Archit. 2008, 54, 707–728. [CrossRef]
17. Park, S.H.; Min, G.I.; Lee, H.J.; Kim, S.W. Increasing Pipeline Stages of the EISC Processor. In Proceedings of the 7th International Conference on Green and Human Information Technology (ICGHIT ’19), Kuala Lumpur, Malaysia, 16–18 January 2019; pp. 89–94.
18. Lee, H.; Beckett, P.; Appelbe, B. High-performance Extendable Instruction Set Computing. In Proceedings of the 6th Australasian Conference on Computer Systems Architecture (ACSAC ’01), Gold Coast, Australia, 29–30 January 2001; pp. 89–94.
19. MIPS Technologies. MIPS 32-bit ISA Manual. Available online: https://inst.eecs.berkeley.edu/~cs61c/resources/MIPS_Green_Sheet.pdf (accessed on 27 March 2019).
20. Berkeley. RISC-V 32-bit Base ISA Manual. Available online: https://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf (accessed on 27 March 2019).

21. Xilinx. 7 Series FPGAs Data Sheet. Available online: https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf (accessed on 19 December 2018).

22. Samsung. Samsung 65 nm Process Technology. Available online: https://www.samsungfoundry.com/foundry/homepage/anonymous/technology12inch65nm.do?_mainLayOut=homepageLayout&menuIndex=020105 (accessed on 19 December 2018).

23. Synopsys. Design Compiler: RTL Synthesis. Available online: https://www.synopsys.com/support/training/rtl-synthesis/design-compiler-rtl-synthesis.html (accessed on 19 December 2018).

© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).