Instruction level redundant number computations for fast data intensive processing in asynchronous processors

Jeong-Gun Lee a,*, Euiseok Kim b, Dong-Ik Lee a

a Department of Information and Communications, Kwang-Ju Institute of Science and Technology, 1 Oryong-dong, Puk-gu, Kwang-Ju 500-712, South Korea
b Samsung Advanced Institute of Technology, South Korea

Received 2 June 2002; received in revised form 27 November 2003; accepted 29 September 2004
Available online 7 January 2005

Abstract

Instruction level parallelism (ILP) is strictly limited by various dependencies. In particular, data dependency is a major performance bottleneck of data intensive applications. In this paper we address acceleration of the execution of instruction codes serialized by data dependencies. We propose a new computer architecture supporting a redundant number computation at the instruction level. To design and implement the scheme, an extended data-path and additional instructions are also proposed. The architectural exploitation of instruction level redundant number computations (IL-RNC) makes it possible to eliminate carry propagations. As a result execution of instructions which are serialized due to inherent data dependencies is accelerated. Simulations have been performed with data intensive processing benchmarks and the proposed architecture shows about a 1.2–1.35 fold speedup over a conventional counterpart. The proposed architecture model can be used effectively for data intensive processing in a microprocessor, a digital signal processor and a multimedia processor.
© 2004 Elsevier B.V. All rights reserved.

Keywords: Redundant number computation; Microprocessor architecture; Asynchronous circuits

1. Introduction

In its short lifetime of 26 years, microprocessors have achieved total performance growth of greater than 10,000 fold [17]. Moreover, industries plan to achieve 100 BIPS by 2010 through the integration of one billion transistors on a single chip. Though rapid advances in semiconductor processing and
computer architecture technologies support a high degree of parallelism, the concurrency of an instruction code is seriously limited by various dependencies [20]. Recently, some new architectures using a billion transistors have been suggested for resolving these dependencies [9,18].

In this paper, we propose a new architecture exploiting a redundant number computation (RNC) at an instruction level (e.g., architectural level) in order to accelerate data computation which is strictly serialized due to data dependencies. Even though the RNCs, which are well-known as carry-save computations, are frequently used in fast arithmetic units and ASICs [8], it is unique and original to use the RNCs at the architectural level, and to adopt several number representations to the data path and the instruction set of a microprocessor [11]. This is an orthogonal approach to previous studies, which tried to speculate on the dependencies in order to reduce pipeline stalls. The speculations are a branch prediction for control dependencies and a value prediction [13] for data dependencies.

In the instruction level redundant number computations (IL-RNC), each IL-RNC based functional unit works with a given input data set and produces a pair of data such as (carry, sum) in the form of conventional redundant numbers rather than a single value. Namely, the computation is partially performed unless the result is strictly required in the type of a single value. In order to enjoy the notion of the IL-RNC, architectural supports over a datapath and instruction set are devised. In addition, asynchronous design techniques are adopted in order to fully utilize the fast processing of the IL-RNC based functional units and eliminate the design difficulties imposed by a global clock. Incorporating the notion of the IL-RNC and the asynchronous design to current high-performance architectures leads to about 1.2–1.35 fold speedup on some data intensive benchmarks.

This paper is organized as follows; Section 2 presents preliminaries necessary for better understanding of this paper. In Section 3, the extended datapaths and instructions for the IL-RNC are explained in detail. In Section 4, the advantages of implementing the proposed architecture on an asynchronous design method is presented. In Section 5, experimental results are given to show the effectiveness of the proposed architecture, and finally, the conclusions are presented in Section 6.

2. Preliminaries

2.1. Data dependency relations

Data dependencies are classified as one of the following three types; read after write (RAW), write after write (WAW) and write after read (WAR), according to the execution order of read and write instructions. Three types of data dependencies cause performance degradation, due to the pipeline stalls in general pipelined microarchitectures. In contemporary high-performance computer architectures, WAR and WAW can be avoided by register renaming while RAW is still difficult to solve. Therefore, RAW is considered as a true data dependency. To resolve the RAW dependencies, several techniques such as value prediction have been proposed [13].

In particular, special emphasis is given to the iterative structures such as for or while structures. In those iterative structures, the RAW dependencies may create very long instruction sequences being composed of repetitive time critical instruction chains, which should be executed sequentially. These long instruction sequences seriously limit the performance of a system. Therefore, efficient data processing in loop structures is an important issue for improving the system performance.

2.2. Carry-save adder structure

When an addition of three or more operands is performed using a two-operand adder, the carry-propagation, which is time-consuming, is repeated in each addition. If the number of operands is k, the carry-propagation occurs \((k - 1)\) times. Several techniques for the multiple operands addition with less carry-propagation penalty have been proposed and implemented. One of the most well-known is carry-save addition [10]. In carry-save
addition, a carry is propagated only in the last step, while in all other steps a partial sum and a sequence of carries are generated separately. A carry-save adder (CSA) for $n$-bit three operands, $A$, $B$, and $C$, with a carry-in, $c_0$, is shown in Fig. 1. Note that a CSA is called a (3,2) counter or a 3-to-2 reduction circuit because it receives three operands and generates two partial results. Therefore, CSAs constitute a datapath to add multiple operands without carry-propagation. The structure of a (3,2) counter can be extended to accept more inputs with only small additional delay.

2.3. Asynchronous system architecture

Asynchronous circuits avoid the use of a global clock, which imposes several serious limitations such as performance and power consumption [6]. Asynchronous circuits which are operating under a localized handshaking protocol may improve system performance by exploiting a locally optimized timing for each functional unit. Although the handshaking of asynchronous local control circuits imposes overhead in terms of performance, a recent research [19] on the reduction of the handshake overhead seems promising.

3. Architectural extensions for IL-RNC

In this section, extended datapaths and an instruction set for an IL-RNC are explained in detail. An addition, a subtraction, a multiplication, and a shift are considered as functional units supporting the IL-RNC in this paper. Here, we explain only an addition and a multiplication since a subtraction and a shift are implemented in a similar fashion.

3.1. Basics of IL-RNC

As explained previously, operations are not processed completely in the redundant number computation flow. Instead, partial results are yielded in the form of redundant numbers which are (carry, sum) pairs, and these pairs are used as input data of the following operations. A non-redundant number representation of the results also can be obtained at any point of the computation flow from redundant number representations. It is worthwhile to recognize that all of these are performed at the instruction level. Notice also that the IL-RNC is different from conventional RNC used mainly in arithmetic circuits or ASICs. The conventional RNC has been used only in order to optimize the circuits which perform given specific functions [8,10], while the IL-RNC allows the fast RNC to be used for various application programs by instruction scheduling.

In the rest of this paper, we use the term redundant value (RV) for a partially processed pair of data which is represented in the redundant number form and use the term non-redundant value (NV) for a completely processed single datum. Fig. 2 shows the benefits of the IL-RNC in the aspect of performance roughly. In Fig. 2, an addition, a multiplication and an addition are linearly ordered because of a RAW data dependency. The processing time of a conventional computation flow is

![Fig. 1. Carry-save adder: (3,2) counter.](image_url)

![Fig. 2. Benefits of IL-RNC.](image_url)
longer than that of the IL-RNC flow since the functional units of the IL-RNC do not perform operations completely. It is worthwhile to note that the proposed IL-RNC is not a dependency resolution method such as a value prediction technique, but the method of suppressing the computation and reducing the time of operations.

Fig. 3 shows a simple view of the architecture that supports the IL-RNC. In addition to the conventional functional units, in order to support the IL-RNC, some extra functional units such as (3, 2) counter and (4, 2) counter are augmented.

3.2. Addition for redundant values

Table 1 shows the extended instructions for the IL-RNC. An addition with redundant values has four types. First, the instruction 32C adds three non-redundant values and generates a pair of data, i.e., an RV. The functional unit is simply implemented by a CSA. Thus, it can also perform an addition of one NV and one RV then produces an RV. Similarly, the instruction 42C takes two RVs as inputs, executes the addition and finally produces an RV. This instruction is a typical addition instruction for two RVs. Instructions 52C and 62C are the extension of 32C and 42C for code optimizations. When two 42C are executed sequentially, combining these two instructions into one 62C reduces processing time and code size. The additions done by 32C and 42C instructions are called redundant additions. The redundant additions can be executed without carry-propagation and the corresponding processing delays are independent of the bit width of data. A subtraction for a redundant computation is implemented in a similar way. Thus, they can be integrated into the same addition unit.

To show the real performance advantage of a redundant addition, the completion time of sequentially ordered three instructions shown in Fig. 2 is calculated with a 64-bit width data. A conventional fast adder (e.g., carry lookahead adder) takes \( \log_2(\text{data-width}) \) full adder delays (FADs) approximately in the worst case. Therefore, in the case of 64 bit data, the adder takes six FADs. Assume that both multiplications shown in Fig. 2 take the same delay, denoted by \( D_\times \). Since an adder for the IL-RNC, a (4, 2) counter, takes only two FADs, the total delay of the three ordered instructions is \( 2 \text{ FADs} + D_\times + 2 \text{ FADs} \) in the case of the IL-RNC while conventional computation requires \( 6 \text{ FADs} + D_\times + 6 \text{ FADs} \).

### Table 1

<table>
<thead>
<tr>
<th>Operation</th>
<th>Instruction</th>
<th>Delay</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>32C</td>
<td>1 FAD</td>
<td>NV + RV → RV</td>
</tr>
<tr>
<td>Addition</td>
<td>42C</td>
<td>2 FADs</td>
<td>RV + RV → RV</td>
</tr>
<tr>
<td>Addition</td>
<td>52C</td>
<td>3 FADs</td>
<td>NV + RV + RV → RV</td>
</tr>
<tr>
<td>Addition</td>
<td>62C</td>
<td>3 FADs</td>
<td>RV + RV + RV → RV</td>
</tr>
<tr>
<td>Multiplication</td>
<td>2MUL</td>
<td>–</td>
<td>NV × NV → RV</td>
</tr>
<tr>
<td>Multiplication</td>
<td>3MUL</td>
<td>–</td>
<td>NV × RV → RV</td>
</tr>
<tr>
<td>Multiplication</td>
<td>4MUL</td>
<td>–</td>
<td>RV × RV → RV</td>
</tr>
</tbody>
</table>

- NV stands for non-redundant value;
- RV stands for redundant value;
- FAD is 1-bit Full Adder Delay;
- Delay marked by “–” will be given in the Section 3.3.
FADs”. In this simple comparison, the benefit from the IL-RNC is clearly observable and it can be said that the gain is coming from the fast redundant additions. Furthermore, $D_\times$ itself is also reduced in the IL-RNC as explained in the following subsection.

### 3.3. Multiplication for redundant values

In this section, a redundant multiplier, which is a multiplier supporting the IL-RNC, is presented. A tree multiplier is considered as a redundant multiplier since it is near to an optimal multiplier structure in terms of processing time. Redundant multipliers can be classified as one of the four following types based on combinations of the input and output value types.

- $NV \times NV = NV$ (Type NNN): This is a conventional multiplier and its worst case delay is the sum of tree depth and a final carry-propagation delay.
- $NV \times NV = RV$ (Type NNR): In this type, the worst-case delay is delay of the tree logic. Since a result of the NNR type multiplication is a redundant value, a final carry-propagating addition is not needed as shown in Fig. 4.
- $NV \times RV = RV$ (Type NRR): Let $a$ be a non-redundant value and $b$ be a redundant value represented by a pair of data $(b_1, b_2)$. The result of $a \times (b_1, b_2)$ can be expressed as $(a \times b_1, a \times b_2)$ by distributive law. To make an NRR type multiplier, two NNR type multipliers are allocated for $a \times b_1$ and $a \times b_2$. Since these two NNR type multipliers produce four values, the final result can be obtained in the form of a redundant value using a $(4,2)$ counter. Compared to the NNR type multiplier, the NRR type multiplier additionally takes 2 FADs in the final $(4,2)$ counter (see top of Fig. 5).
- $RV \times RV = RV$ (Type RRR): The RRR type multiplier is composed of two NRR type multipliers and a $(4,2)$ counter as shown in bottom of Fig. 5. The $(4,2)$ counter is used in order to reduce two redundant values, which are generated from two NRR type multipliers, into one redundant value. Thus, the delay of the RRR type multiplier is two FADs longer than that of the NRR type multiplier.

Until now, the four types of tree multipliers are introduced and analyzed in the viewpoint of delays. The processing times of the four tree multipliers are compared.

<table>
<thead>
<tr>
<th>Multiplier type</th>
<th>Worst case delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>(instruction code)</td>
<td>32 bit</td>
</tr>
<tr>
<td>NNN (MUL)</td>
<td>$8^a + 6^b = 14$</td>
</tr>
<tr>
<td>NNR (22MUL)</td>
<td>$8$</td>
</tr>
<tr>
<td>NRR (32MUL)</td>
<td>$8^a + 2^c = 10$</td>
</tr>
<tr>
<td>RRR (42MUL)</td>
<td>$8^a + 4^c = 12$</td>
</tr>
</tbody>
</table>

- $a$: Tree reduction depth.
- $b$: Final carry-propagation delay.
- $c$: Counter delay for generating an RV result.

![Fig. 4. Multiplier structure for computing $NV \times NV = RV$ and its symbol.](image)

![Fig. 5. Multiplier structures for computing $NV \times RV = RV$ and $RV \times RV = RV$.](image)
multiplier types are summarized in Table 2, where the unit delay is a single full adder delay. The redundant multipliers are triggered by the corresponding instructions shown in Table 1. This speedup of the redundant multipliers is achieved at the cost of many transistors and which seem to be available sufficiently in the near future [17,18]. The area of an NRR type multiplier is about twice as great as that of an NNR type multiplier while the area of an RRR type multiplier is also about twice as great than that of an NRR multiplier.

In addition to conventional functional units such as an ALU and a multiplier, the RNC based functional units are added to a datapath to support IL-RNC. The area increase due to these functional units is determined by the type and the number of each added functional unit. The area of the IL-RNC based functional units can be approximated using the areas of a counter logic and a tree reduction circuit. When $n$-bit data width is assumed, a $(3, 2)$ counter and a $(4, 2)$ counter use $n$ and $2n$ 1-bit full adders respectively. For a redundant multiplier, a tree reduction logic consists of $n^2$ AND gates, $n^2 - 4n + 3$ 1-bit full adders, $n - 1$ 1-bit half adders when Dadda’s method is used [4].

3.4. Loop extension

Efficient computation of a loop structure is a critical issue on the performance improvement in data intensive applications because loop iterations increase the length of a critical path considerably. In order to accelerate the execution of the iterating critical path, the application of IL-RNC to a loop structure is addressed in this subsection.

Fig. 6 shows a typical program code of a loop computation and the corresponding data flow graph. The loop code iterates one hundred times. The loop computation contains a complex and mutually dependent operation between $u_1$ and $y_1$. The principle of code scheduling for the IL-RNC is allocating the IL-RNC functional units to the operations on a critical path. Allocation of IL-RNC functional units is performed iteratively until no more delay reduction of critical path delay is achieved. For the sake of proper code scheduling, the instruction code generation consists of three parts: head computation, body computation and tail computation as shown in Fig. 7.

### 3.4.1. Head computation

See the variables $u$ and $y$ in Fig. 7. They are used to calculate the next values of themselves. If the variables are not available in the form of an RV at an entry of the loop, then the variables have different forms at the inputs ($u$ and $y$) and outputs ($u_1$ and $y_1$) since the $u_1$ and $y_1$ are calculated in a redundant number form. Due to the different representation forms of the variables, the code for the first iteration cannot be used iteratively. The purpose of the head computation is deriving redundant values from the non-redundant values during the first loop-iteration step. The derived redundant values are then used as input data for the body computation that is performed iteratively. If all the variables used in the loop body are available in the form of redundant values, then there is no need to do the head computation.

For example, in Fig. 7, consider the input data $x$, $dx$, $u$ and $y$. At first, since $dx$ is a constant, $dx$ remains as a non-redundant value throughout the loop processing. We know that the addition calculating $x_1$ in the data flow graph (DFG) in Fig. 6(b) is not on the critical path. Thus the addition for $x_1$ does not need to be performed in the IL-RNC. Finally, $u$ and $y$ are used as input data to calculate redundant values, $uu$ and $yy$, which are used iteratively in the body computation. From the DFG in Fig. 7, the code for the head computation is generated as follows:
32CSUB and 42CSUB are subtraction instructions for the IL-RNC. The performance of this example is seriously limited by the critical path $\oplus_2 \rightarrow \oplus_5 \rightarrow \oplus_7 \rightarrow \oplus_9$, where each subscript corresponds to the node number in the DFG of Fig. 7. Furthermore, since the operation $\oplus_2$ uses the output of operation $\oplus_9$ in the loop, the length of serial instruction sequence significantly increases according to the number of loop repetitions. Here, this type of instruction chains is referred to a “loop cycled critical instruction chain” (LCCIC). To reduce the computation time of the LCCIC, the IL-RNC instructions are scheduled to the operations on the LCCIC. The total latency of the LCCIC is 46 ($= 17 + 17 + 6 + 6$) FADs with a conventional architecture and 25 ($= 10 + 12 + 1 + 2$) FADs with the architecture supporting the IL-RNC.

### 3.4.2. Body computation

Using the redundant values that are obtained during the head computation phase, the code which is executing iteratively is constructed. The following code is the body computation code for the example in Fig. 7.

```plaintext
; Body Computation Code
; x->R1, dx->R4, uu->(R18,R19), yy->(R11,R12)

LD R30, #99 ;R30 is used for INDEX "i"

BODY_LOOP: ; 99 times repeat

; Node Insn.
1  ADD R5, R4, R1 ;x+dx = R1
2  32MUL (R18,R19), R4, (R6,R7);u*dx = (R6,R7)
3  32MUL (R11,R12), R4, (R8,R9);y*dx = (R8,R9)
4  42MUL (R18,R19), (R11,R12), (R20,R21);u*y = (R20,R21)
5  32MUL R1, (R6,R7), (R22,R23);dx*node-2's result = (R22,R23)
6  42C (R6,R7), (R11,R12), (R14,R15);node-2's result + y
7  42CSUB (R22,R23), (R18,R19), (R24,R25);node-5's result - u
8  42CSUB (R14,R15), (R20,R21), (R11,R12);compute "y1" in a RNC
9  42C (R24,R25), (R8,R9), (R18,R19);compute "u1" in a RNC
MOV R1, R5 ;R1 -> R5
SUBBNZ R30, #1, R30, BODY_LOOP
```
The differences between the head and the body computation codes stem from the availability of a redundant value and an extra loop control instruction such as a SUBBNZ, where SUBBNZ is an instruction integrating a “subtraction” and “branch if non-zero”. Note that the total number of the code repetitions is 99 since the first execution has been done in the head computation.

3.4.3. Tail computation

Tail computation transforms the redundant values generated from the body computation into non-redundant values. For optimization purpose, tail computation codes can be avoided by directly applying the redundant values to the head computation of the next code block. The following code is the tail computation code generated from the source code in Fig. 6.

```assembly
; Tail Computation Code
ADD R11, R12, R26 ;Transform RV to NV for y1
ADD R18, R19, R27 ;Transform RV to NV for u1
```

3.5. Miscellaneous

Up to now, only arithmetic instructions for IL-RNC have been considered. In order to apply the notion of the IL-RNC to processor architectures more effectively, a register indexing method and a branch condition check scheme for redundant values are discussed.

3.5.1. Register indexing

One problem in IL-RNC is the large size of register indexing bits because the IL-RNC instructions require many operands. Those instructions make an instruction word design very difficult in real implementations. In addition, instruction decoding may take long time. To reduce the number of register indexing fields and to make the instruction format more consistent with a conventional instruction format, a new indexing technique for redundant values has been devised using an auxiliary register file. Each register in an auxiliary register file has a corresponding register in the main register file. The main register file is normally used for a non-redundant value. When a redundant value is required, however, both the main and the auxiliary register files are used together. To distinguish between these two cases, a special bit must be appended to each register-indexing field. If the bit is set, the register index is sent to both the register files, and a data pair implying a redundant value is read out as shown in Fig. 8.

3.5.2. Fast branch condition resolution

In a branch instruction, a redundant value should be transformed into a non-redundant value.
when the redundant value is compared with a certain non-redundant value. Because the transformations involve a carry-propagation, branch instructions may cause performance degradation in loop processing. Early work on a fast branch condition resolution were done by Cortadella [2] and Lutz [14,15]. Their work present methods to evaluate a condition such as \(a_1 + a_2 = b\) quickly. In their methods, evaluation is performed without any real addition or subtraction requiring carry-propagation. Therefore comparisons between a redundant value \((a_1,a_2)\) and a non-redundant value \(b\) can be done by checking only \(a_1 + a_2 = b\) without any data transformation.

4. The impact of asynchrony on IL-RNC

To maximize the effectiveness of the proposed IL-RNC architecture, asynchronous designs are considered as an underlying design method. In this section, the necessities and advantages for implementing the concept of the IL-RNC with an asynchronous methodology are presented.

In terms of delay, the IL-RNC architecture is characterized as followings:

- some functional units supporting the IL-RNC have comparatively short execution time,
- the delay variation is very high among the functional units.

Considering these two features, an asynchronous design technique has the following potential advantages:

- **Overall performance enhancement from the locally optimized processing delays of functional units:** Due to the delay variation among the functional units, it is hard to enjoy the fast operations of IL-RNC functional units with the global clock based synchronous design style. In general, “16 FO4 delays” is selected as a clock cycle time and “8 FO4 delays” is considered as the limitation of a feasible clock cycle [1,7]. Assume a synchronous system whose clock period is 12 FO4 delays. The clock period includes clocking overhead delays and computation delays. If clocking overhead takes about 3 or 4 FO4 delays, then 8 or 9 FO4 delays are used for computations. Even though some computations finish earlier than 8 or 9 FO4 delays (redundant additions for 32C need only 2 FO4 delays), their results cannot be used until the next clock edge and the corresponding functional units cannot be allocated to other new instructions. Therefore, benefits of the fast computation cannot be exploited. In consequence, the fast processing of the IL-RNC architectures cannot be achieved in synchronous designs. On the other hand, asynchronous systems have an inherent feature that each functional unit can be locally optimized in the aspect of the processing time through a local handshake signaling. As soon as the computations produce results, they can be used directly without the global clock based synchronizations. Therefore, asynchronous system design style is an effective way to integrate the functional units in the IL-RNC architecture.

- **Easy pipeline stage partitioning, less pipeline overhead:** The fast clock cycling demanded by the IL-RNC architecture may cause a critical deep pipeline problem when synchronous design technique is adopted. If the clock cycle time is set to about 6 FO4 delays (2 FO4 computation delays + 4 FO4 delays for a clocking overhead), a conventional tree multiplier demanding 17 FADs should be pipelined into almost 17 stages in a synchronous system. In this situation, due to the latch timing overhead, this deep pipelining increases the latency of the functional units and may diminish
the performance gains of the IL-RNC significantly. Note that the timing overhead for latch-based designs becomes a prohibitive fraction of the clock cycle when the system runs faster than 16 FO4 delays per clock [1,7]. In asynchronous designs, the functional units can be partitioned into any number of pipeline stages without considering global clock time constraints. Consequently, latency increases in the pipelined functional units can be avoided in asynchronous designs.

- **No clock skew caused by fast clock cycle:** As explained above, in order to fully take advantage of the faster processing time of the IL-RNC functional units, the clock cycle time should be set to near to one FAD (2 FO4 delays) if synchronous design techniques are adopted. In this case, with a 4 FO4 delay clocking overhead, the clock frequency reaches near to 2 GHz (5 or 6 FO4 = "3 or 4 FO4 for a clocking overhead" + "2 FO4 for a full adder delay") with a 0.18-µm technology or higher frequency with a deep sub-micron technology. Clock skew and on-chip synchronization become serious problems in this situation. In an asynchronous design, however, the high speed clock distribution problem is not a concern.

The above three facts show that an asynchronous design method is indispensable in order to guarantee the performance gain of the IL-RNC. The only problem is handshake overhead of asynchronous circuits, but the overhead has been significantly reduced to roughly 666 ps (about 3–4 FO4 delays) with a 0.35-µm technology in [19] recently and this delay overhead seems to be comparable to the synchronous clocking overhead. Therefore, synchronous implementation of the IL-RNC is not considered in this paper and it is assumed that the IL-RNC is implemented in an asynchronous design method.

5. Performance evaluation

In this paper, we suggest a computer architecture supporting the IL-RNC with new instructions and the corresponding functional units. The proposed method accelerates execution time effectively for a sequence of instruction codes serialized by data dependencies. In order to show the performance effects of the proposed architecture, we perform the following two simulations. Firstly, we give simulation results for simple scalable examples to show potential performance advantages of the IL-RNC. Secondly, well-known practical data intensive applications are used in the simulation to show the real effectiveness. Since the synchronous implementation of the proposed architecture cannot exploit the benefit of the IL-RNC due to its cycle time limit as mentioned, we do not consider the synchronous case.

5.1. Evaluation architecture model

For the simulations, we have developed an asynchronous superscalar architecture simulator shown in Fig. 9 with C++ [12]. A superscalar architecture is selected since it is one of the most widely used high-performance computer architectures, and the performance can be improved by integrating IL-RNC instructions and the corresponding functional units.

Evaluation architecture models implemented on the asynchronous superscalar architecture simulator have the following features: In a Fetch/Decode unit shown in Fig. 9, up to 4 instructions can be fetched when a request signal comes from a RF Read/Rename unit. No instruction cache miss and perfect branch prediction are assumed. Since the loops in data intensive applications have many iterations and have good memory locality in general, the assumptions are reasonable. In the RF Read/Rename unit in Fig. 9, we substitute operands of each instruction by data or tags, and the renamed instructions are sent to an Issue unit. Up to 20 instructions can wait in the rename buffer of the RF Read/Rename unit. Issue and Reorder Buffer units have 20 and 60 instruction buffer slots, respectively. In order to manage multiple asynchronous functional units without metastability,
handshaking and arbitration are used in the asynchronous architecture and their behaviors are modeled using event-driven simulation. Finally, the data cache is assumed to always hit. This is a reasonable assumption in data intensive application programs since the data distribution of those programs has good locality on a memory block.

In our simulation, two types of evaluation architecture models are implemented. The first is the reference architecture model (RA) which is a conventional asynchronous superscalar architecture as described above. The other is the proposed architecture model (PA) which is the extension of the reference model with the new instructions and the corresponding functional units. In experiments, various processing latencies of the components in Fig. 9 are assumed in order to investigate the sensitivity of the proposed scheme over various architectural overheads. The architectural overheads are classified into the following four levels over four delay parameters;

*Levels of Architectural Overhead*

- Level-0 delay setting; FETCH DELAY = 0.1 FAD, RENAME DELAY = 0.1 FAD, ISSUE DELAY = 0.1 FAD, CACHE MEMORY DELAY = 0.1 FAD, etc.
- Level-1 delay setting; FETCH DELAY = 3 FADs, RENAME DELAY = 3 FADs, ISSUE DELAY = 3 FADs, CACHE MEMORY DELAY = 3 FADs, etc.
- Level-2 delay setting; FETCH DELAY = 4 FADs, RENAME DELAY = 4 FADs, ISSUE DELAY = 4 FADs, CACHE MEMORY DELAY = 4 FADs, etc.
- Level-3 delay setting; FETCH DELAY = 6 FADs, RENAME DELAY = 6 FADs, ISSUE DELAY = 6 FADs, CACHE MEMORY DELAY = 6 FADs, etc.

The processing delays of functional units in the Execution unit are listed in Tables 1 and 2.

5.2. Evaluation results for simple scalable codes

Scalable instruction codes are composed of only addition and multiplication instructions. The instruction sequences consist of conventional ADD and MUL instructions for RA. For PA, 42C and 42MUL instructions are used instead. Four different mixing ratios of additions to multiplications are considered. These instructions are totally ordered by RAW data dependencies so that the instructions have to be executed one by one with the results of the previous instructions.

Fig. 10 shows the speedup achieved by the IL-RNC with various mixing ratios of two simple instructions according to the four architectural overhead levels. Four performance indexing curves are presented for the corresponding four instruction mixing ratios.

The data processing under the Level-0 environment can be considered as a nearly data flow computation where only function processing delays are taken into account. The speedup in the Level-0 is near to the upper bound of the performance improvement which can be achieved by the IL-RNC for the given instruction sequences. As the level of architectural overhead increases, the speedup is reduced since the delay of non-enhanced parts (Fetch/Decode, RF Read/Rename, Issue, Reorder Buffer, and RF Write units) increases. Consequently in order to maximize the effect of the IL-RNC, it is important to reduce the latencies of the non-enhanced components shown in Fig. 9. The Level-3 overhead mode seems to be a general case (1.34 speedup in average). We expect that the latencies of those components can be reduced to a certain extent by pipelining in order to reduce the architectural overhead more.
Independent of the simulation results, a simple formula approximating the speedup is roughly derived. The ratio of addition to multiplication instructions is given as \( \alpha : \beta \) and the variable \( \delta \) denotes the value corresponding to the level of architectural overhead. Approximate speedup formula is expressed as

\[
\frac{\alpha' \times \text{Delay}(\text{ADD}) + \beta' \times \text{Delay}(\text{MUL}) + \delta}{\alpha' \times \text{Delay}(42\text{C}) + \beta' \times \text{Delay}(42\text{MUL}) + \delta},
\]

where the function \( \text{Delay}(\text{inst}) \) returns the processing delay of the functional unit issued by the instruction \( \text{inst} \), \( \alpha' \) is \( \frac{\alpha}{\alpha + \beta} \) and \( \beta' \) is \( \frac{\beta}{\alpha + \beta} \). The formula approximates the speedup found through the performance evaluations closely when the variable \( \delta \) is set to the maximum latency among the latencies of the non-enhanced components shown in Fig. 9. From the similarity between the simulation results and analytic results, the validity of our simulation can be justified to a certain degree.

5.3. Evaluation results for practical codes

To give more practical view, the example program fragment shown in Fig. 6 and other three well-known program codes, a differential equation solver, a time consuming inner loop code of a Mandelbrot image generation program and an IIR filter, are used as benchmarks. All the benchmarks are not only loop intensive but also data-processing intensive. Two instruction codes are manually generated from the high-level descriptions of those benchmarks for RA and PA. In order to investigate the effectiveness of the proposed IL-RNC under various operating conditions about some resource limited cases, performance evaluations are performed under the following four architecture configurations.

**Architecture Configuration Classes**

- **Configuration 0**: sufficient functional units are available and all the functional units are not pipelined. Here, Level-0 mode is used for an architectural overhead. This configuration is used for both RA and PA.
- **Configuration 1**: same as Configuration 0 except for the level of architectural overhead. In this configuration, instead of Level-0, Level-3 mode is used. This configuration is used for both RA and PA.
- **Configuration 2**: For PA, two multipliers for each redundant multiplier type (NNR, NRR, RRR) are used and the redundant multipliers are pipelined into two stages. For RA, no resource limitation and the use of non-pipelined functional units are assumed. Level-3 mode is used for both PA and RA.
- **Configuration 3**: For PA, only one multiplier for each redundant multiplier type is used and the redundant multipliers are pipelined into two stages. For RA, no resource limitation and the use of non-pipelined functional units are assumed. Level-3 mode is used for both PA and RA.

The benchmark instruction codes are processed on the simulator which is properly configured according to the above four configuration classes. Notice that PA has less functional units than RA in the Configurations 2 and 3. In addition, since no resource limitation is assumed in RA, there is no need to make pipelined functional units having increased latency due to latch overheads. Here, the pipeline latch overhead is assumed to 1 FAD. Therefore, this assumption is advantageous to RA. The evaluation results for the benchmarks simulations are shown in Fig. 11.

Except for the Mandelbrot inner loop code execution in Configuration 3, all other benchmarks show about a 1.2–1.35 fold speedup. For the Mandelbrot inner loop code, three multiplications are

\[
\text{Average Speedup} : 1.81
\]

\[
\text{Average Speedup} : 1.325
\]

\[
\text{Average Speedup} : 1.23
\]

\[
\text{Average Speedup} : 1.177
\]

**Example Code**

- **IIR Code**
- **Mandelbrot Inner Loop Code**
- **Differential Equation Code**

![Performance Comparison for Practical Examples](image-url)

Fig. 11. Performance comparison for four practical examples.
executed concurrently with each other. Furthermore, they are all on time-critical long instruction chains. Since all the multiplications are on the time-critical instruction chains, the slack time (in other words, mobility [3]) for each multiplication becomes almost zero. In consequence delaying any of the multiplications, due to the lack of available functional units, increases the processing time of the instruction chains directly. No performance gain is observed in the simulation in that case. Therefore, under the condition of limiting functional resources, RA may have better performance than PA by implementing more NNN type multipliers instead of a large-size redundant multiplier. However in the other three cases even though the resources for redundant values are limited and pipelined, performance gains are still preserved. Furthermore, resource limitation would not be serious in near future, because advances in semiconductor technology are expected to provide sufficient transistors according to the 2000 SIA roadmap [17].

Comparison of reorder buffer utilizations provided in Table 3 are obtained from the simulation using Configuration 1. The results show that the maximum number of allocated reorder buffer entries is smaller than that of a conventional computation counterpart. This means that the blocking rate in a reorder buffer is lower in the IL-RNC superscalar architecture. The reason for this is that the faster processing of IL-RNC functional units allows the corresponding instructions to retire from the reorder buffer without blocking the following instructions too greatly. Consequently, with comparatively smaller reorder buffer, performance benefits are achieved. From the point of performance, less reorder buffer can be an important design factor since a result-forwarding logic in the reorder buffer requires high control overhead and the logic may cause a delay penalty [5].

Currently, it is under investigation to decouple “RV to NV transformations” from the execution stage. This can be done by allowing the transformations to be done at the reorder buffer between the completion and retire of instructions. This decoupling may eliminate later issue of “RV to NV transformation instructions” and improve the performance further.

6. Conclusions and future work

In this paper, a computer architecture supporting IL-RNC is proposed to accelerate the processing of long instruction chains, which are sequentially ordered by RAW data dependencies. Compared to the reference architecture, the suggested architecture has faster functional units. Furthermore, to effectively exploit the various and fast processing delays of the functional units, an asynchronous design methodology is adopted as an underlying design methodology. Finally to show the performance benefits of the proposed architecture, performance evaluations have been done and a 1.2–1.35 fold speedup is observed. The proposed architecture is expected to be used effectively in the data intensive processing such as digital signal processing or multimedia processing.

Future work is investigation of code optimization in the suggested architecture for better performance. In addition, a hardware-sharing method is being considered for the high utilization of redundant multipliers. Since the circuit structures of the redundant multipliers are very similar to each other, the hardware-sharing can be implemented easily with the speculative completion delay [16] according to the type of redundant multipliers.

Table 3
Reorder buffer utilization comparison

<table>
<thead>
<tr>
<th>Benchmark program</th>
<th>Avg. alloc. ROB entries</th>
<th>MAX alloc. ROB entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>RA:PA</td>
<td>RA:PA</td>
<td></td>
</tr>
<tr>
<td>Example code</td>
<td>19.5:13.9</td>
<td>28:26</td>
</tr>
<tr>
<td>DiffEq</td>
<td>21.7:18.5</td>
<td>30:27</td>
</tr>
<tr>
<td>Mandelbrot-IL</td>
<td>19.5:10.5</td>
<td>25:18</td>
</tr>
<tr>
<td>IIR Filter</td>
<td>31.3:8.6</td>
<td>41:14</td>
</tr>
</tbody>
</table>

References