UNIVERSITY OF CALIFORNIA,

IRVINE

Effect of High-Radix Carry-Save Redundancy in ALU on Processor Cycle Time, Architecture, and Implementation

THESIS

submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in Electrical and Computer Engineering

by

Fernando R. Negre

Thesis Committee:

Professor Tomas Lang, Chair

Professor Douglas Blough

Professor Fadi Kurdahi

1999

The thesis of Fernando Negre is approved:

____

____

____
Committee Chair

University of California, Irvine

1999

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I want to thank the Balsells family, the Generalitat de Catalunya and the University of California for their support through the Balsells Fellowship, and Professor Roger H Rangel, coordinator of the Balsells Fellowship program, for his invaluable help.

This work has also been funded in part by the National Science Foundation Grant MIP-9726244.

I would also like to thank Professor Lang for his support and guidance, and Professor Blough and Professor Kurdahi for being on my committee.

Thanks to my Balsells fellows, especially Alba, to the Spaniard postdocs: Tonio, Susana, Esther, Encarni and Carmen, to Paco the saxophonist, to Rafa and M. Jose and to my friend Lili. Also thanks to my supportive roommates Long, Hoan, Du, Wesley and Kevin.

Finally, the people who helped with the nonacademic issues were John, Loretta, Linda and Ronnie. Their work has been very important for me.

ABSTRACT OF THE THESIS

Effect of High-Radix Carry-Save Redundancy in ALU on Processor Cycle Time,

Architecture, and Implementation

By

Fernando Negre

M.S. in Electrical and Computer Engineering

University of California, Irvine, 1999

Professor Tomas Lang, Chair

There is a trend in computing towards bigger programs. As program size increases so does the addressing space, therefore, it is required that the data word width be expanded. In terms of CPU performance, the main problem with this approach is the ALU cycle time. The addition operation requires the carry to propagate through all the data-word bits, as currently implemented.

The carry propagation chain determines the critical path, which in turn determines the cycle time. It is desired to shorten the cycle time.

In this thesis, the method proposed to shorten the cycle time is cutting the ALU adder carry-propagation chain. Thus, one level of gates is saved if conditional-sum or look-ahead tree-like (i.e. logarithmic) schemes are used.

The problems that arise when cutting the adder carry-propagation chain are examined, discussed, evaluated and solutions are developed and proposed to solve them. The redundant architecture proposed is illustrated by discussing the modifications necessary to an existing pipelined CPU.

EFFECT OF HIGH-RADIX CARRY-SAVE REDUNDANCY IN ALU ON PROCESSOR CYCLE TIME, ARCHITECTURE, AND IMPLEMENTATION

I. INTRODUCTION AND OBJECTIVE

1.1 Introduction

There is a trend in computing towards bigger programs. As program size increases so does the addressing space, therefore, it is required that the data word width be expanded. In terms of CPU performance, the main problem with this approach is the ALU cycle time. The addition operation requires the carry to propagate through all the data-word bits, as currently implemented. In other places where additions are present, the same problem can be found, but those additions are considered later as they are less likely to become a problem.

In the following, we assume that the critical path in a pipelined processor is in the ALU or execution stage. The carry propagation chain determines the critical path, which in turn determines the cycle time. It is desired to shorten the cycle time, and the method chosen to do this is cutting the ALU adder carry-propagation chain by half (Figure 1). Thus, one level of gates is saved if conditional-sum or look-ahead tree-like (i.e. logarithmic) schemes are used.

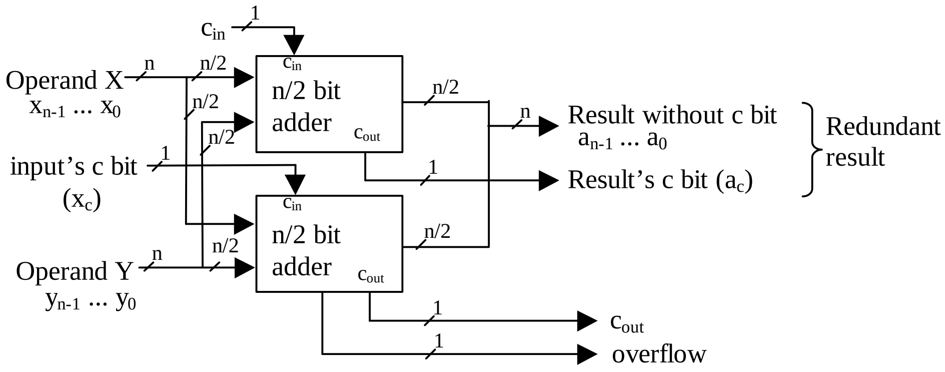


Figure 1: Proposed redundant adder and redundant representation (at its output). The broken carry propagation chain is shown. The result requires n+1 bits. Note that two conventional operands can be added, or one redundant with one conventional, but two redundant operands cannot be added using this configuration, since there is only one c bit input.

As a consequence of this technique used to save time, some of the instructions performed by the ALU/execution stage (i.e. those involving the adder) will give a result in redundant representation instead of conventional representation. The redundant result, with its unadded c-bit, cannot be used as it is in some cases, but it can be used in some other cases. Most of this thesis is about which are these cases, what to do when the result cannot be used in its redundant form, and what are the implications on performance. The conversion to conventional representation, when necessary, will be done in other stages.

Of course, the carry chain can be cut in 3 or more parts instead of just two. These other possibilities are further investigated. In general, the n word bits are divided in m-bit groups. There are n/m such groups so n/m adders are used. For each group there is a carry-bit that has to be stored. The resulting notation is called radix-2^m carry-save redundant representation.

The method of dividing in just 2 parts introduces the minimum amount of redundancy possible, i.e. just one bit, and also the minimum amount of changes to any preexisting implementation. It is the most convenient when the critical path time of the ALU is only slightly larger than the other stages' times. This is the most likely situation to happen while increasing the address space, since any current processor pipeline has well-balanced stages that will be unbalanced. The execution stage time is increased more than any other stage time.

Dividing the adder in half leaves a not propagated carry bit, as was shown in Figure 1. The carry bit has to be saved, and it later has to be used together with the other bits in subsequent operations, or to convert the redundant result to a conventional representation if that is required. The represented value is

$$\text{value} = \left(\sum_{i=0}^{n-1} a_i \, 2^{i} \right) + a_c \, 2^{n/2} \qquad (1)$$

where a_i is the i^{th} bit of the n bit value already present in the conventional representation and a_c is a new bit with the same weight as $a_{n/2}$. Note that, if d_i were the bits representing the same value in conventional (natural binary) representation, then $a_i=d_i$ for $i<n/2$, and the equality would only hold for $i \geq n/2$ when $a_c=0$.

The operations that produce a redundant result are addition and subtraction. An illustration of the redundant representation used is shown in Figure 2.

$$\begin{array}{cccccccccc}
 & x_{n-1} & x_{n-2} & \cdot & x_{n/2+1} & x_{n/2} & x_{n/2-1} & \cdot & x_1 & x_0 \\
 & & & & & x_c & & & & \\
+ & y_{n-1} & y_{n-2} & \cdot & y_{n/2+1} & y_{n/2} & y_{n/2-1} & \cdot & y_1 & y_0 \\
\hline
 & a_{n-1} & a_{n-2} & \cdot & a_{n/2+1} & a_{n/2} & a_{n/2-1} & \cdot & a_1 & a_0 \\
 & & & & & a_c & & & &
\end{array}$$

Figure 2. Addition in Radix $2^{n/2}$ carry-save redundant representation. Note that the digits, $a_{n/2}$ and a_c, $x_{n/2}$ and x_c, and $y_{n/2}$, have weight $2^{n/2}$.

The block diagram of the module that converts from radix $2^{n/2}$ to conventional representation, called the 'conditional incrementer', is as shown in Figure 3.

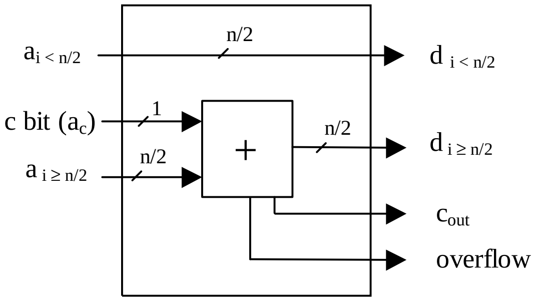


Figure 3. Block diagram of a conditional incrementer used to convert a value from radix-$2^{n/2}$ carry-save redundant representation into conventional representation. Some gate-level implementations of this module are discussed in the "Conversion Circuit" chapter.
The module is called 'conditional incrementer' since it increments a's upper half depending on the value of the c bit.

1.2 Objective

The objective of this thesis is to investigate the possibility of using redundancy in a scalar processor to reduce addition times. It is desired to know what problems are introduced by doing so. Solutions have to be developed for these problems. The importance of any problem that cannot be solved has to be evaluated.

II. RADIX $2^{n/2}$ BASIC PIPELINE IMPLEMENTATION

2.1 Original pipeline organization

The MIPS microprocessor has been chosen as the basis to try the redundant implementation due to its simple instruction set architecture. Since new dependencies between instructions arise while using redundancy, it is convenient to use a RISC processor to keep the amount of those new dependencies to a minimum. In this section, an overview of the processor is given. In the next section, the modifications are described.

Figure 4 shows a simplified view of an implementation of a MIPS processor. The proposed modifications are shown in Figure 5. Details of the basic MIPS R2000 5-stage pipeline implementation that has been used here can be found in [4]. The 5-stage pipeline has been chosen because it is the simplest. However, the principles applied to modify it in order to use redundant representation are directly applicable to more recent and longer MIPS pipelines.

The pipeline is designed for fast execution of a reduced instruction set that only makes use of direct, indirect, immediate and offset addressing modes. The basic instructions are addition, subtraction, ‘and’, ‘or’, shift, move, store, load, ‘branch on [not] equal’ and ‘set if less than’ (SLT).

The move-from-register-to-register instruction is not an instruction on its own. It is a particular case of the addition where one of the addends is zero.

The ‘set-if-less-than’ instruction compares two registers, or a register and an immediate, and assigns a one to the destination register when the condition holds, zero otherwise.

The ‘branch on less than’ doesn’t exist. It has to be built by composing a ‘set if less than’ with a ‘branch on equal’ that checks the previous result with the desired logical value. ‘Set on equal’ doesn’t exist either, but can be built using a ‘branch on not equal’ put in between two assignment instructions.

The original implementation shown in Fig 4 consists of 5 pipeline stages: Instruction Fetch, Instruction Decode, Execute, Memory and Write Back. They are noted using the letters are F, D, E, M, W, respectively. Between each pair of consecutive stages, there is a latch. F/D refers to the latch between the Fetch and the Decode stages. The other latches are noted in the same manner.

In the first stage (Instruction Fetch) there is the Program Counter (PC) that points to the following instruction to be read from the Instruction Cache. The logic that increments the PC is present in the same stage.

The second stage contains the Register File and the Branch logic. The Register File has two read and one write ports. An equality checker controls the branching multiplexer in the Instruction Fetch stage (control lines are not shown). A stall is needed when branching has to be decided on the result of the previous instruction, since the branch decision is taken in the Decode stage.

The third stage is the Execution stage. It consists of the ALU and multiplexers that select the operands. There are some forwarding paths leading to this stage that reduce the impact of data dependencies on performance.

The memory stage consists of the data cache. Data that has to be written to the Register File skips the memory system and goes directly to the E/M latch. A stall is needed after a load if using the loaded value in the next instruction.

Finally, the Write Back stage selects between the value read from the Data Cache and the one coming from the ALU in order to write it to the Register File.

2.2 Pipeline modifications

The modifications needed for the pipeline for the redundant implementation are shown in Figure 5. The modifications are explained below, stage by stage.

- The Instruction Fetch stage doesn’t need any change. It includes an addition, but since its overhead is much lower than the Execution Stage overhead, it is not likely that this addition will be in the critical path. The Exexution Stage overhead is the time spent to direct the operands (multiplexers in Fig. 6), plus the time to invert the

bits when 2’s complementing one of the operands (XOR gate in Fig. 7), plus the time to select between the result of the adder or that of the logic unit or the shifter (mux in Fig. 7).

In the case the IF adder is in the critical path, maybe because a bigger amount of redundancy is used in the Execution Stage, radix-$2^{n/2}$ redundancy can be used for the ‘next instruction’ addition as explained in the 5th chapter.

- The equality checker at the Decode stage operates on conventional operands and requires a conversion, so a converter is included in the forwarding path. Alternatively, an equality checker that accepts a redundant input, like the one shown in section “5.6 Redundant CAM”, can be used. The redundant equality checker does shorten the comparison time as compared to converter plus conventional checker.
- A conversion is also necessary when forwarding the ALU result back to its input when the operation to be performed doesn’t accept a redundantly represented operand as input (which operations are these ones will be shown later). For those operations that accept a redundant representation a new forwarding path (FWDR) is provided. While forwarding the ALU result back to the ALU input for a subsequent addition the value doesn’t need to be converted.

The forwarding multiplexers are shown in Figure 6.

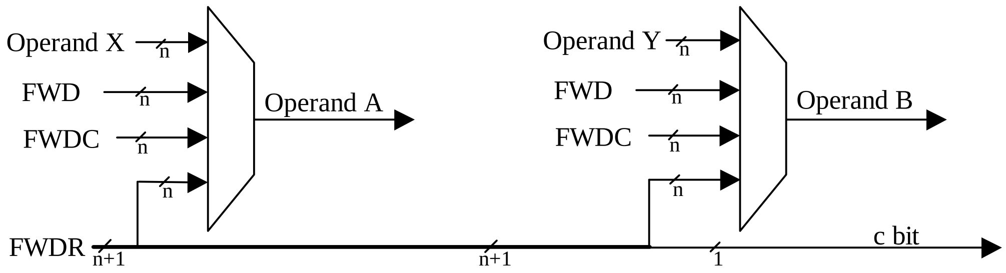


Figure 6: Multiplexers that implement the forwarding paths to the ALU.

- The memory stage needs a redundant to conventional representation converter as well, since the ALU result is in redundant representation depending on the operation performed. The conversion has to be performed before feeding the value as a data memory address. (In fact, the conversion can be avoided at some expense, as explained in section “5.5 Conversion time reduction for memory stage”).
- Finally, no change is required for the Write Back stage. Any value arriving here has already been converted in the memory stage.

2.3 ALU implementation

The ALU implementation is shown in Figure 7. It is based on the adder shown in Fig 1. It includes no equality checker since equality is checked for in the Decode stage.

Some control signals are shown in the figure. ‘FWDR used’ is asserted when the multiplexers of Figure 6 select the FWDR bus. It allows the c-bit to be fed to the adder. ‘adder used’ is asserted when an addition or subtraction or SLT is performed by the ALU. It ensures that the resulting c-bit is 0 when it is not being used, so that the interpretation of the result as a redundant value is also correct.

The ‘CMP’ signal is asserted during subtraction and SLT comparison, to obtain the 2’s complement of operand B.

The ‘NEG’ signal is asserted to obtain the 1’s complement of operand B, which is necessary for avoiding some stalls when executing a SLT in certain cases, as explained in section “2.10 SLT instruction”.

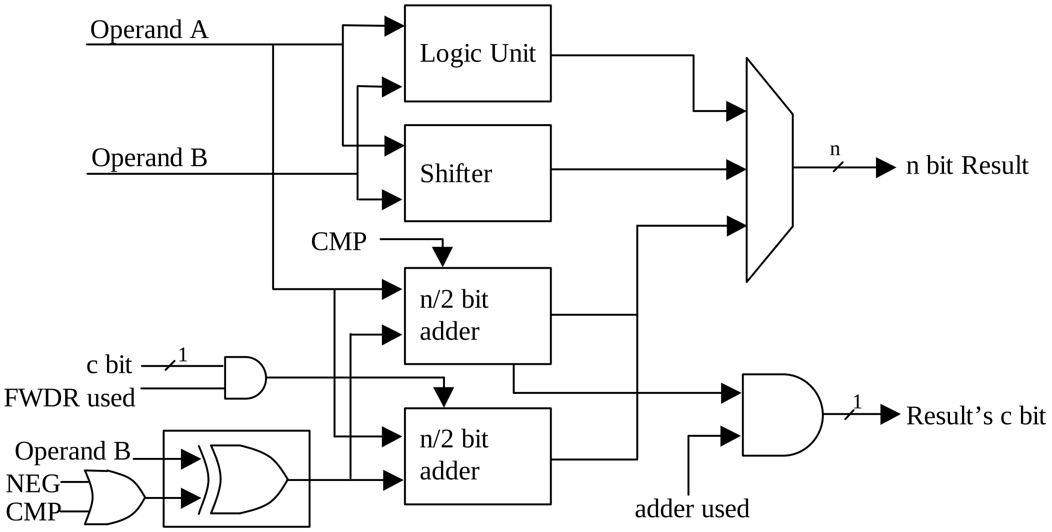


Figure 7: Proposed redundant ALU (execution stage). The shifter is the only non-commutative module. Let us assume that the 1st operand is to be shifted by the amount of the 2nd. The CMP control signal 2's-complements Operand B. NEG is required for SLT inputs swap only and 1's-complements Operand B (see section “2.10 SLT instruction”).

2.4 Conventional, redundant and converted operands

The multiplexers in Fig. 6 determine the origin of the operands. Some restrictions may apply when performing some operations depending on the origin of the operands. In the following three sections, the representation-dependency-caused restrictions are analyzed. Three terms are defined here in order to do the analysis.

- Conventional: Operand in conventional representation that is taken either directly from the register file (Operands X, Y, as in Fig 5, 6), or forwarded from the M/W latch through FWD.
- Redundant: Operand in radix-$2^{n/2}$ carry-save representation. This is the previous ALU result forwarded using FWDR.
- Converted: The representation is conventional, but the value is not available until after the conversion delay time (see section "4.5 Incrementer delay"). That an operation is able to take a converted operand means that it takes a short time, in the sense that there is time to fit it into one cycle together with the conversion. This point is illustrated in the next section. Converted operands come through FWDC (Converted Forward, as shown in figures 5 and 6).

Summary: Conventional and redundant values have different representations but are available at the beginning of the cycle. Converted and conventional values have the same representation but are available at different times during the cycle.

2.5 Representation dependencies

When using the proposed implementation, representation-dependencies appear in addition to the usual control and data dependencies.

Representation dependencies are defined as data dependencies that involve a value that is in a representation that does not suit the needs of the operation that has to be performed on it. The ideal case in which no dependency arises is shown in Figure 8a.

The new dependencies arise from two facts:

1. Some operations cannot operate on redundant values. Thus, they need a previous conversion of the operands if any of them is in redundant representation. This conversion can be done in the same cycle as the operation, and we call the resultant value a ‘converted operand.’ In other cases, it can take its own cycle so that the operation can be done on a ‘conventional operand’.

2. One of the operations (SLT) requires a final conversion of an intermediate redundant value before giving its final result. This is because the carry has to be propagated in order to know the sign, which determines the operation outcome. The conversion will be done during the second cycle. In case there is time for both, the conversion and the next operation are both performed during that second cycle. In case there is not enough time for both, the operation is performed one cycle later.

The next figures show the new situations produced by operations that produce a redundant result or produce a converted one. Conventional results do not introduce a new situation, so they are not considered in this section.

In the following examples, ‘short operation’ denotes an operation that can be accomodated in one cycle with a conversion. A ‘long operation’ requires more time and cannot be accomodated together with the conversion. The next section deals with this issue in detail.

- Case 1: An operation leaves a result in redundant representation. The following operation is data dependent on the result but can accept it in redundant representation.

Example: (note the data dependency on r1) (shown in Fig. 8a)

```
add r1, r2, r3
add r4, r1, r5
```

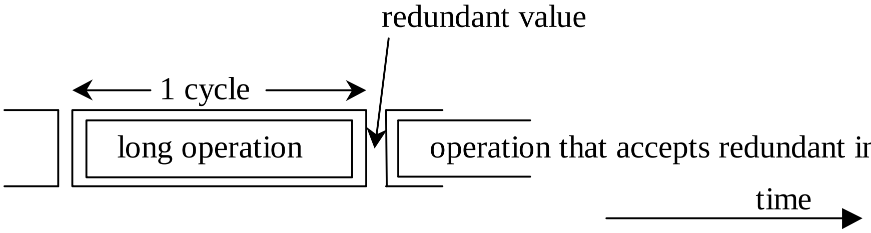


Figure 8a: Both cycles take place in the execution stage. The situation in where an operation that provides a redundant result is followed by another operation that accepts a redundant operand is shown.

The operations that can be found in the situation of the second operation above are marked in Table 1 as able to take a redundant operand as input. Table 2 shows which operations produce a redundant result.

- Case 2: An operation leaves a result in redundant representation. An operation which is data-dependent on it follows. The latter cannot take a redundant operand, but can wait for it to be converted, and lasts a short enough time to be performed in the same cycle as the conversion. This is illustrated in Fig. 8b.

Example: (note the data dependency on r1) add r1, r2, r3

and r4, r1, r5

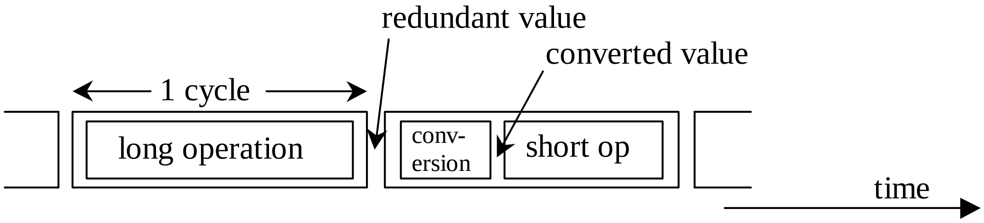


Figure 8b: All cycles are ALU cycles. The situation in where an operation that provides a redundant result is followed by another operation that accepts a converted operand is shown.

The operations that can be found in the situation of the 'short-op' above are marked in Table 1 as able to accept converted values as input.

- Case 3: This case is illustrated in Fig 8c. An operation produces a redundant result. The operation that follows is data dependent on it. The latter operation cannot take a redundant operand and is not short enough to be performed in the same cycle as the conversion of the value.

Example: (note the data dependency on r1, which is the subtrahend of the 2nd instruction)

add r1, r2, r3

sub r4, r5, r1

A whole cycle is devoted to the conversion. The conversion is counted as a ‘stall’, since the subtraction has to be stopped until the operand it needs is converted and forwarded.

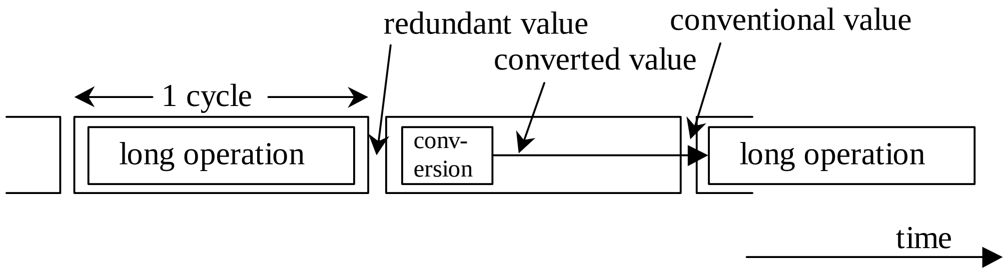


Figure 8c: Both 'long operation' cycles take place in the execution stage. The conversion takes place in the memory stage. The situation in where an operation that provides a redundant result is followed by another operation that accepts neither a converted operand nor a redundant one is shown.

The operations that can be found in the situation of the second 'long-op' above are marked in Table 1 as not able to accept neither converted nor redundant values.

- Case 4: The 'Set if Less Than' (SLT) instruction needs a conversion to be always performed right afterwards, regardless the next instruction. If the next instruction is not data dependent on the result of the first one, no stall is required, since the conversion can be done in the memory stage in parallel with the next instruction. If the next instruction is data dependent and short enough, it can be fitted in the same cycle in which the conversion is performed, as was done in case 2 above. Then the conversion and the next instruction are both performed in the execution stage.

Example: (Shown in Figure 8d. Note the data dependency on r1)

```
slt r1, r2, r3

and r4, r5, r1
```

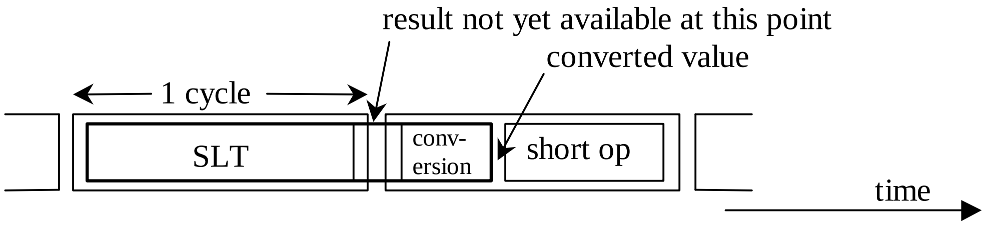


Figure 8d: Both SLT and (conversion + ‘short-operation’) cycles take place in the execution stage. The situation in where an operation that provides a converted result is followed by another operation that accepts a converted operand is shown.

The operations that can be found in the situation of the 'short-op' above are marked in Table 1 as being able to accept converted values as input.

- Case 5: Same as case 4, but the second instruction, data dependent on the first one, does not fit right after the conversion, so that it has to wait until the next cycle, when the conversion has finished.

Example: It is illustrated in Figure 8e, (note the data dependency on r1).

```
slt r1, r2, r3

add r4, r5, r1
```

In the situation represented in Figure 8e, the conversion is counted as a ‘stall’, since the second operation (an ‘add’ in the example) has to be stopped until the operand it needs is converted and forwarded.

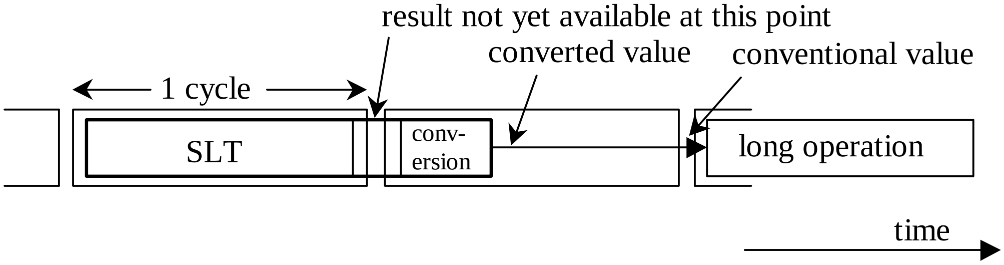


Figure 8e: The situation in where an operation that provides a converted result is followed by another operation that accepts only a conventional operand or a redundant one is shown. Both SLT and 'long-operation' cycles take place in the execution stage. The conversion that precedes the 'long operation' takes place in the memory stage.

The operations that can be found in the situation of the second 'long-operation' above are marked in Table 1 as not able to accept neither converted nor redundant values.

2.6 Input representation dependencies

Table 1 summarizes the dependence of each operation on the representation of its operands. Here we see what problems can arise when trying to immediately forward a redundant result back to the ALU input.

Logic operations can take conventional operands, but cannot take any redundant operand. This means that the carry has to be propagated prior to performing the logic operation. They can take converted operands too. This means that in case an operand is in redundant representation there is enough time available to make the conversion during the same cycle. This is so since logic operations take a short time as compared to addition and subtraction.

Operations that use the adder are considered arithmetic. They are additions, subtractions and set-if-less-than (SLT). The three of them can take conventional operands but none of them can take converted operands, since they are in the critical path and so there is no time for the conversion. However, in general they can take redundant operands, so the conversion is not necessary. The exception is the subtrahend (it is subtraction's and SLT's case) that cannot be redundant since it has to be 2's-complemented.

2.7 Redundant 2's complement

The process of 2's-complementing a redundant number is the same as for a conventional one. The number has to be subtracted from 2^n.

For conventional numbers, this operation is reduced to inverting all the bits and adding one (since inverting all the bits is equivalent to performing '2^n-1-number'). However, this shortcut is not useful in redundant representation, due to the c-bit. When the c-bit is one a subtraction of $2^{n/2}$ is needed as a correction. This subtraction is equivalent to an addition of $c\cdot(2^n-2^{n/2})$ (c bit's 2's complement). $2^n-2^{n/2}$ looks like this: 11...1100...00 (n/2 ones followed by n/2 zeroes), and the addition has a maximum carry-chain of length n/2 bits.

The conclusion is that to change the sign a conditional decrement has to be performed on the upper half of the operand, which has at least the same complexity of the conditional increment discussed previously. Therefore, 2's-complement of a redundant number will not be done directly. Conversion will be performed previously, instead.

2.8 Input representation dependencies for branches, loads and stores

In addition to the ALU operations, branches, loads and stores are considered. A branch takes as operands two registers and an immediate destination displacement. It is conditional on equality of the operands. Equality can be checked in a short time compared with addition time. Thus, the operands it can take can be conventional or converted. In addition, as shown in [6], the comparison can also be done in redundant representation. The approach taken here is converting before comparing.

In case there were not enough time to write the operands into the register file and read and compare them, an additional forwarding path could be added so that writing and comparing are performed at the same time.

Stores have two operands: a data word and an address. In the implementation considered, the memory always takes a conventional data word and a converted address value. Loads have only an address formed by one operand plus an immediate offset. In the implementation considered in this chapter, the address will always be a converted value. The fact that converted values can be taken is not a granted property of stores but a design requirement. The necessity of performing a conversion in the memory stage can affect cycle time and thus performance. Section “3.6 Redundant architecture application example” deals with this issue.

		representation of 1st and 2nd operands				
instruction		conventional, conventional	conventional, redundant	redundant, conventional	conventional, converted	converted, conventional
logic	AND	yes	no	no	yes	yes
	OR	yes	no	no	yes	yes
	XOR	yes	no	no	yes	yes
shifts (1)	variable	yes	yes (2)	no	enough time? (3)	enough time? (3)
	immediate	yes	-	no	-	enough time? (3)
add		yes	yes	yes	no	no
subtract (4)		yes	**no** (5)	yes	**no** (5)	no
check if equal (6)		yes	yes	yes	yes	yes
set if less than (4)		yes	**no** (5)	yes	**no** (5)	no
branch if equal		yes	yes	yes	yes	yes
store word (7)		yes	no	no	yes	yes

	input representation		
	conventional	redundant	converted
load word (8)	yes	no	yes

(1) Shift: 1st operand is value to be shifted. 2nd operand is the shifting amount.
(2) There is no problem in specifying the amount of shifting in radix $2^{n/2}$ representation since only the $(1+\log_2 n)$ least significant bits matter.
(3) It is unclear whether there is enough time for a conversion plus shift to be performed. It can depend on the particular realization. It is at first assumed that there is enough time, later it is shown that the impact of the assumption not being true is low.
(4) Subtraction and SLT: 1st operand is minuend, 2nd operand is subtrahend.
(5) Subtractions and SLT with a redundant or converted subtrahend cannot be performed. Note that all the other operations can be done on either a redundant or a converted representation. That is, either the redundant operand can be taken or there is enough time to convert.
(6) This is part of 'branch on equal' and is performed in the 'decode' stage.
(7) Store: 1st operand is value to be stored. 2nd operand is address.
(8) Load: Only one operand (address).

Table 1: MIPS instructions and influence of non-converted conventional (coming from the Register file or FWD), redundant (coming from FWDR) and converted (coming from FWDC) operands. This table shows what kind of operands can each operation take.

2.9 Two redundant operands

Table 1 doesn't consider the case of an operation with two redundant or two converted operands, since only one value can be in redundant representation at a given time.

This situation occurs only when both operands are the same. However, it is relatively common.

The possible cases are:

1. add ry, rx, rx
2. sub ry, rx, rx
3. slt ry, rx, rx
4. beq rx, rx, destination

Since our redundant adder doesn’t accept two redundant inputs at the same time, the approach taken is to change the operations above to equivalent ones that avoid the problem. Respectively

1. sll ry, rx, 1; shift register x one position to the left and store in ry
2. add ry, r0, r0; clear register y
3. add ry, r0, r0; clear register y
4. j destination; absolute jump

The change of op-code and operations will take some time in the decoding stage. However, since only the first case is likely to ever occur, the 2nd and 3rd cases can be detected afterwards and a bubble can be used to allow the operands to be converted, just to ensure correctness. In the first case, the operand change cannot be avoided. Furthermore, checking the overflow has to be added to the shift, so that in this case the behavior of the instruction is kept.

2.10 SLT instruction

The subtraction instruction cannot be performed on a redundant subtrahend, since the negation operation required to perform a two's complement requires conventional representation. Since the SLT instruction subtracts the operands, and checks the sign in order to verify the inequality, in principle this instruction cannot be performed when the operand to be the subtrahend is in redundant representation. However, it can be chosen which one operand to complement. The two possibilities are shown below (where the '+' sign means arithmetic addition, and '' stands for one's complement.)

$$a<b \Rightarrow a-b<0 \Rightarrow a+b'+1<0$$

$$a<b \Rightarrow b-a>0 \Rightarrow b-a-1\geq 0 \Rightarrow b+a'\geq 0$$

Therefore, the algorithm for the SLT instruction can be, in order to avoid representation-caused stalls:

```
if (a is redundant)
        $result_0$ = sign of $(a+b'+1)$
otherwise [b can be redundant]
        $result_0$ = complemented sign of $(b+a')$,
```

where $result_0$ is the least significant bit of the result. All the other result bits are 0.

Therefore, the SLT operation consists in the following sequence

1. bit-inverting ($a \leftarrow a'$) or negating ($b \leftarrow b'+1$) the operand in conventional representation, then

2. adding with the other operand, and then,

3. converting, which takes place during the following cycle.

The conversion conditionally increments the result's upper half, as usually, in order to check its sign (as shown in section "4.6 Complete conversion circuit").

2.11 Representation of result and operations' latency

The ALU result representation depends on the operation performed. It is redundant for additions and subtractions. It is conventional for all others. Though, for inequality comparison instructions (SLT) the conventional result is available only after conversion of the adder redundant output, and, as stated before, this is called a 'converted' value to remark that fact.

Table 2 details the availability time of the result and its representation.

ALU operations		representation of result	result availability time
logic	AND	conventional	end of cycle
	OR	conventional	end of cycle
	XOR	conventional	end of cycle
shifts		conventional	end of cycle (1)
add		**redundant**	end of cycle
subtract		**redundant**	end of cycle
check if equal (2)		conventional	end of cycle
set if less than		converted (3)	**next cycle, after conversion**

(1) If there is enough time for the shift to be performed.

(2) This operation is part of 'branch if equal' in the MIPS architecture. It is performed in the 'decode' stage.

(3) That a value is 'converted' implies that it is not available at the end of the cycle, as is stated in the "Conventional, Redundant and Converted operands" section.

Table 2: Availability and representation of result for each ALU operation class

Note the ‘Set if Less Than’ instruction that produces a ‘converted result’. Every operation will provide the result by the end of the cycle, be it either in conventional or redundant representation, except the SLT instruction. That a result is ‘converted’ means that it is provided in conventional representation, but it is available after conversion, and not at the beginning of the cycle.

The SLT instruction requires always a conversion to conventional representation so that the sign bit of the result can be obtained and checked (As is shown in section “4.6 Complete conversion circuit”).

The conversion may be avoided at some cost making use of the logic circuits described in [5].

2.12 Stalls

When an operation cannot be performed on neither redundant nor converted operands, then a stall will be needed if one of those operands is in redundant representation. An operand can only be in redundant representation when it is (1) The result of last cycle’s ALU computation (i.e. there is a data dependency between two consecutive instructions), and (2) the operation then performed produced a result in redundant representation (i.e. it was an addition or subtraction).

When the SLT operation is followed by an operation that depends on its result and cannot take a converted operand (i.e. additions, subtractions and SLT itself), a stall is required as well.

The operations that would require a stall when one input is redundant are subtraction and ‘set less than’, but only in the case that the operand that has to be subtracted (subtrahend) is in redundant representation. However, in a SLT operation minuend and subtrahend can always be switched (as discussed in section “2.10 SLT instruction”) so that only subtractions require stalls.

Table 3 summarizes all the actions that have to be taken depending on the representation dependency. It has been built using the information shown on Tables 1 and 2.

instruction	**result availability time and representation** (3)	**next instruction**	**requirement for next instruction**	**action**
add or subtract	redundant	add, minuend of sub, SLT	redundant or conventional	none
		subtrahend of sub	conventional	**stall** (1)
		logic, shift	conventional or converted	convert
		branch on equal	conventional or converted	convert
all except add, sub, SLT	conventional	add, minuend of sub, SLT	redundant or conventional	none
		subtrahend of sub	conventional	none
		logic, shift	conventional or converted	none
		branch on equal	conventional or converted	none
SLT	converted	add, minuend of sub, SLT	redundant or conventional	**stall** (2)
		subtrahend of sub	conventional	**stall** (2)
		logic, shift	conventional or converted	none
		branch on equal	conventional or converted	none

(1) Situation represented in Fig 8c.

(2) Situation represented in Fig 8e.

(3) That a value is ‘converted’ implies the time at which it is available during the cycle as well as its representation. Refer to the “Conventional, Redundant and Converted operands” section (2.4) for details.

Table 3: Action to take depending on the representation dependency

2.13 Stall control

It can be said, as a summary of the “Stalls” section, that a stall is needed when the result of an inequality comparison (SLT or similar) is to be added or subtracted (or again SLTed) by the next instruction. (Table 3, note (2))

A stall is also needed when the result of an addition or subtraction is the subtrahend of the next instruction (and this is a subtraction). (Table 3, note (1))

No more stalls are necessary for the changes proposed to the implementation considered. The stalls necessary in the original architecture, as explained in section “2.1 Original pipeline implementation”, still are so. These stalls are still required in the new implementation. The following describes how to treat the new situations.

To determine whether a stall is required or not, the control logic must look for a SLT instruction in the E/M latch. If its destination register is used as source of an addition or subtraction in the D/E latch a stall is required. It should also look for subtractions at D/E latch where the subtrahend is the destination register of an immediately previous (E/M latch) addition or subtraction. In both cases, a bubble should be dynamically inserted. Then the converted result will be taken from the M/W latch through FWD. The following code shows the new conditions that lead to a stall.

```
1.  if   $opcode_{E/M}$=SLT and ($opcode_{D/E}$=ADD or $opcode_{D/E}$=SUB or
         $opcode_{D/E}$=SLT) and ($operand1_{D/E}$=$dest_{E/M}$ or $operand2_{D/E}$=$dest_{E/M}$)
then stall
2. if    ($opcode_{E/M}$=ADD or $opcode_{E/M}$=SUB) and $opcode_{D/E}$=SUB
          and $operand2_{D/E}$=$dest_{E/M}$
```

then stall.

A stall consists in

1. stalling the Program Counter update, the Instruction Fetch and Decode stages (i.e. not latching the stage results into the PC, F/D and D/E latches),
2. inserting a bubble in the Execution stage (E/M latch), and letting the Memory stage (M/W latch), that performs the conversion, proceed.

III. PERFORMANCE EVALUATION

3.1 Performance evaluation

The assumptions are that the addition takes only a fraction of the original cycle time, and that the rest, the overhead time, remains unchanged.

The speed up, or performance improvement, is computed using the formula,

$$\text{improvement} = 1- \frac{T_{NE}\,(1+\text{stall}_N)}{T_{OE}\,(1+\text{stall}_O)} \qquad (2)$$

Which assumes that the execution stage time, (the old, T_{OE}, and the new, T_{NE}) limit both the old and new clock frequencies.

As the stall rates (old, stall$_O$,a nd new, stall$_N$) are variable (since they depend on the techniques used to reduce them) and difficult to assess (since they depend on the particular benchmark run), the improvement without considering the stalls will first be computed, then the stall rates will be considered.

$$\text{improvement}_{\text{without stalls}} = 1 - \frac{T_{NE}}{T_{OE}} \qquad (3)$$

3.2 Evaluation of reduction of cycle time for radix-$2^{n/2}$ addition

Table 4 shows the evaluation of times and improvements for several word widths, assuming the values for $t_{p,g}$, t_{clg}, and t_s stated in the table.

$t_{p,g}$ is the time it takes to generate the initial propagation and generation signals, n is the number of bits in a word, t_{clg} is the time it takes to generate the intermediate propagation and generation signals, and t_s is the time to perform the sum given all the other signals.

α is the ratio $T_{OVERHEAD}/T_{ADDER}$, where T_{ADDER} and $T_{OVERHEAD}$ are the two components of the execution stage time, $T_E=T_{ADDER}+T_{OVERHEAD}$. ($\alpha$ is the overhead, the addition time taken as unity).

The evaluation below is for a multilevel prefix carry lookahead adder, as lookahead is a usual choice for adder implementations and the prefix implementation is convenient to prevent excessive loads when the word width is high. Two more logarithmic adders are examined in the Appendix, but the results are very close to those given below.

Table 4: Evaluation of increase of performance for radix $2^{n/2}$ multilevel PREFIX carry-lookahead adders

					$\alpha=1$		
					cycle time	cycle time	improvement, %
n	lg n	lg n -1	TmaxCLA	TmaxR2n/2	original	improved	(w/o stalls)
64	6	5	22	19	44	41	6.8
128	7	6	25	22	50	47	6.0
256	8	7	28	25	56	53	5.4
512	9	8	31	28	62	59	4.8
1024	10	9	34	31	68	65	4.4
	$t_{p,g}$	t_{clg}	t_s				
	2	2.5	2	levels of gates		times in '# of gate levels'	

Table 4: Evaluation of increase of performance for radix $2^{n/2}$
(continued) multilevel PREFIX carry-lookahead adders

α=0.33			α=3		
cycle time original	cycle time improved	improvement, %	cycle time original	cycle time improved	improvement, % (w/o stalls)
29	26	10.3	88	85	3.4
33	30	9.0	100	97	3.0
37	34	8.1	112	109	2.7
41	38	7.3	124	121	2.4
45	42	6.6	136	133	2.2

3.3 Quantitative evaluation of representation-dependency-caused stall frequency

Table 5b shows the relative importance of the situations presented in Table 1. The following assumptions are made:

1. Independence among instruction types. That is, there is no correlation between the instruction type at position i and the instruction type at position i+1.
2. (a) Instructions depend on the previous one with 50% probability, if a dependency is possible. (b) Then a value of 100% is used instead.
3. When the dependency can be on either one of two operands, it is considered to be half the time on the first, half on the second.
4. The instruction rates are those on Table 5a (from [4])
5. No fast conversion or c-bit checking or prediction is used. (See their corresponding sections for details)

ALU operations		frequencies, %
logic	AND	3
	OR	5
	XOR	1
shifts		4
add		14
subtract		0.5
set if equal		7
set if less than		7

Table 5a: Assumed percentage of ALU instructions, by type, over instruction count. From DLX data in [4].

The frequencies in Table 5b are computed as follows: The first column is the frequency of a non-redundant and non-converted result (everything except additions, subtractions and SLT, thus 100-14-0.5-7= 78.5) by the frequency of the operation as in Table 5a. Second and third columns are half the additions and subtractions ((14-0.5)/2= 6.75) by the frequency of each operation type. The third and fourth are half the SLT (7/2= 3.5) by the frequency of each operation type.

‘Convert’ indicates that the operation cannot be done in a redundant representation but there is enough time to convert. Thus, the corresponding amount has been added in the corresponding ‘converted’ column. ‘swap’ indicates the swapping explained in section ‘SLT instruction’. The corresponding amount has been added to the ‘redundant, conventional’ column.

		operands				
ALU operations		conventional, conventional	conventional, redundant	redundant, conventional	conventional, converted	converted, conventional
logic	AND	2.4	convert	convert	0.32	0.32
	OR	3.9	convert	convert	0.54	0.54
	XOR	0.78	convert	convert	0.11	0.11
shifts		3.1	0.29	convert	0.14	0.43
add		11	1.0	1.0	**0.49**	**0.49**
subtract		0.39	**0.036**	0.036	**0.018**	**0.018**
set if equal		5.5	0.51	0.51	0.25	0.25
set if less than		5.5	swap	1.01	**0.25**	**0.25**

Table 5b: Frequencies for each situation. See text for assumptions and explanation.

Given the assumptions and the data on Table 5b, the expected stall rate due to the changes proposed is half (because of the 2nd assumption) the sum of the numbers in bold face in Table 5b, thus 0.77%.

Note that the influence of the shifts, if there was not enough time to convert and shift is an additional 0.21%.

Assuming all the instructions to depend always on the previous one (100% probability, assumption 2(b)) and shifts to cause a stall, too, the rate would be higher: 2.1% additional stalls over the instruction count.

3.4 Influence of stall rate on performance

The sources of stalls in the old pipeline that are also present in the new one are any use of a register that is being loaded from memory and mispredicted branches. The new stalls are caused by ADD and SUB instructions that follow SLT instructions and by a SUB instruction that use the result of an immediately preceding ADD or SUB as the subtrahend.

From instruction mix data for some SPECint92 programs for the DLX processor (Table 5a), we know that the ratio of adds is around 0.14 over the instruction count, subtractions ratio around 0.005, and compare ratio 0.13 (we will assume half to be SLT, so a 0.07 ratio). In section “3.3 Quantitative Evaluation [...]”, a value of 0.0077 was obtained as expected rate for the new stalls of the new implementation.

On the other hand, stalls due to loads and branches happen from one every 10 to one every 4 cycles (DLX processor data in [4]). Let us take 0.1 and 0.25 as possible stall rates. Then the $(1+\text{stall}_N)/(1+\text{stall}_O)$ ratio ranges from 1.108/1.10 = 1.0073 to 1.258/1.25=1.0064, what means that the results in the tables above diminish in close to 0.7 percent points each.

3.5 Delay of prefix incrementer and conversion time

Conversion to conventional representation has to be done in the ALU stage (but only for some of the operations) and in the memory stage, as shown in Fig 5.

Conversion involves adding the c-bit to the upper half of the redundant value, but also to select the sign in case SLT is the operation performed. See section “4.6 Complete conversion circuit”.

In order to evaluate the overhead introduced by the conversion to conventional representation, the table below shows the proportion of time used by such conversion with respect to the cycle time allowed by the improved adder. The table is only for radix-$2^{n/2}$ carry save representation. It is assumed that the Memory stage has the same overhead as the Execution stage.

Table 6:
Incrementer time, with respect to the prefixCLA cycle time

		α=1		α =0.33		α =3	
n	incr. Time	cycle time, improved	%	cycle time, improved	%	cycle time, improved	%
64	8.5	41	21	26	32	85	10
128	9.9	47	21	30	33	97	10
256	11.3	53	21	34	33	109	10
512	12.7	59	22	38	33	121	10
1024	14.1	65	22	42	33	133	11

3.6 Redundant architecture application example

While the improvement of adding just a single redundant bit in the internal operand representation is limited, it is also important in that it adds flexibility in the implementation of a pipeline. A simple example follows for illustration.

In this example instead of using the 5 stages MIPS R2000 pipeline the 8 stages one of the MIPS R4000 will be used. This is done in order to use somewhat more current numbers for latencies, i.e. take into account that the latency of memory operations is bigger than that of the execution stage. The redundant implementation proposed here is useful to the same extent no matter the number of pipeline stages. The restriction for the proposed implementation to be useful is that the execution stage is limiting the clock rate, and nowadays this can happen only, and happens, when the memory stages are thoroughly pipelined.

In the design of a pipeline, it is estimated that the delay of the 64-bit CLA-based ALU used is 1 time unit and that of the cache is 2.5 time units. This is a reasonable assumption since the Mips R4000 processor [1] has 1 ALU stage and 3 memory stages; Data first, Data second, Tag check. Thus the ratio is approximately 1:3 for that implementation. If we increase the data word from width 32 to 64 bits the ratio is increased. We assume this new ratio, as stated before, to be 1:2.5.

The usual choices, not considering redundancy, are to pipeline the cache in (1st) 3 stages of 1 time unit or (2nd) 2 stages of 1.25. The choice would depend on the additional stalls due to the increased cache latency when pipelining in three stages and technology constraints.

Let's take a look at a third choice and then compare it with the two first ones. The ALU can be made redundant, thus saving 6.8% of the execution stage time (from Table 4, α=1), resulting in a 0.93 ALU time. Then we need an incrementer that adds to the cache stages. The 'cache plus incrementer' time would be (the 21% value comes from Table 6) 2.5 + 0.21 0.93 = 2.7. If we divide this into three stages we have a cycle time of 2.7/3 = 0.9.

Thus the redundant-ALU time of 0.93 is more restrictive than the memory cycle time, so that limits it and the cycle time has to be 0.93, resulting in a (1-0.93)/1= 7% of improvement in performance over the first approach. After taking into account the new stalls the performance would be reduced down to approximately 6%.

The improvement over the second approach is (1.25-0.93)/1.25= 26% gross - additional stalls due to cache latency, that now is 2.79 (three cycles of 0.93 time units) instead of 2.5 (two cycles of 1.25), should be taken into account in this 2nd choice. They will make the figure somewhat lower.

3.7 Application of radix-$2^{n/2}$ to other additions

Other additions performed that could benefit from similar techniques are the branch and the Program Counter increment adders (see Fig 4 or 5, adders in the first two stages). The critical path is not likely to include those additions since the overhead is smaller here than in the execution stage, due to the various multiplexers, the 1-complement circuit (XOR gates) needed for subtraction, and the bigger amount of area needed.

However, radix $2^{n/2}$ redundancy can still be useful for the two aforementioned additions in case they enter the critical path. That can happen when using a lower radix in the ALU or when pipelining it. The application of radix-$2^{n/2}$ redundancy for address additions is discussed in the next chapter.

3.8 Quantitative verification

In order to contrast the quantitative assumptions made in section 3.3, simulations of the SPECint95 benchmarks have been performed. The (arithmetic) average stall rate due to representation dependencies has been determined to be 0.66% for this set of programs. Therefore, the assumptions made represent a worse-that-can-be-expected case.

EFFECT OF HIGH-RADIX CARRY-SAVE REDUNDANCY IN ALU ON PROCESSOR CYCLE TIME, ARCHITECTURE, AND IMPLEMENTATION

IV. CONVERSION CIRCUIT

4.1 Introduction

In this chapter, procedures for conversion between the radix-$2^{n/2}$ carry-save redundant representation and the conventional one are discussed. After that, the implementation of the conversion circuits is shown. Finally, changes in overflow signaling are examined. The radix-$2^{n/2}$ carry-save redundant representation is discussed in Chapter 1.

4.2 Conversion to and from conventional representation

The conversion from redundant to conventional involves the addition of the c-bit to the upper half of the number. Note that the lower half is left unchanged. Note also that the addition of c is just an increment (of the upper half-word) when c=1, and no change at all when c=0.

$$\left(\sum_{i=0}^{n-1} b_i\, 2^i\right) = \left(\sum_{i=0}^{n-1} a_i\, 2^i\right) + c\, 2^{n/2} \tag{4}$$

The equations describing the conversion are derived from the expression of a two operands addition [7]:

$$b_i = a_i \ ; \forall i < \frac{n}{2} \tag{5}$$

$$b_i = a_i \oplus c \cdot \left(\prod_{j=\frac{n}{2}}^{i} a_i \right); \forall i \geq \frac{n}{2} \tag{6}$$

The fact that to convert from redundant to conventional no change but eliminating c is necessary when c=0 could be used to avoid some otherwise necessary stalls. This idea is explained and expanded in sections “5.3 Fast conversion to conventional representation” and “5.4 c-bit prediction”.

On the other hand, the conversion from conventional to redundant is straightforward and only requires the addition of an extra c bit with a value of 0. Therefore, an adder that accepts redundant inputs also does accept conventional ones.

4.3 The incrementer

The incrementer will take a short time to do its function as compared to a regular adder. The critical path for this element consists in a $\log_2(n/2)$ 2-input AND gates followed by an XOR gate, what will provide the most-significant bit. See Figure 9.

The critical path delay will be big enough not to allow feeding its result back again to the adder input. If we did, the gain obtained in cycle time by using redundancy would be lost, since the delay we just eliminated would be reintroduced, plus some overhead.

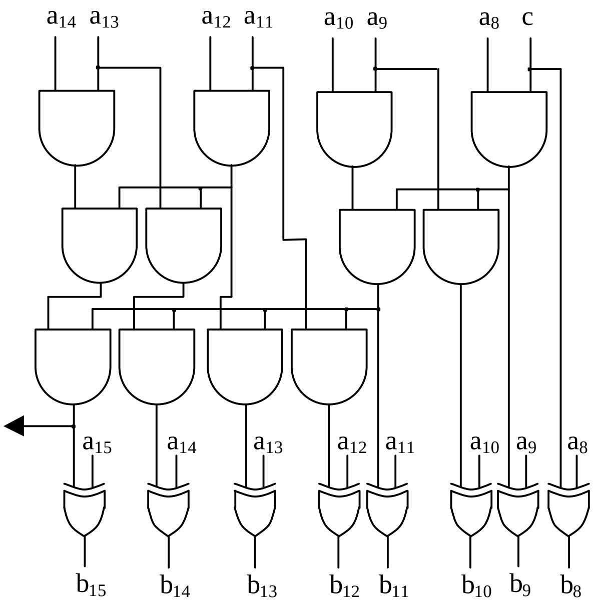


Figure 9. Conditional incrementer for n=16. b=a+c, where c is a single-bit value. It is the same as the expression ‘if(c==1) b=a+1; else b=a;’. Observe the $\log_2$ modularity. The arrow is the output for expanding to a bigger module (n>16).

4.4 Practical implementation for the incrementer and delay

Since NAND and NOR gates are faster than AND gates, the following implementation (Figure 10) is faster than the one in Figure 9.

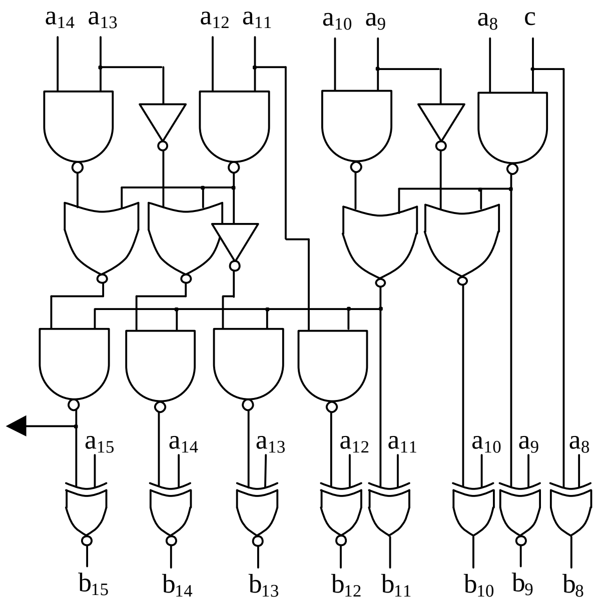

a_{14} a_{13} a_{12} a_{11} a_{10} a_9 a_8 c
a_{15} a_{14} a_{13} a_{12} a_{11} a_{10} a_9 a_8
b_{15} b_{14} b_{13} b_{12} b_{11} b_{10} b_9 b_8

Figure 10. Conditional incrementer. Practical implementation using fast gates.

Let us derive a general delay expression for the module. Let the delay of 2-input NAND and NOR gates be the same linear function of the number of loads L (d_{NGATE} = a + b L). And let d_{XOR} be the delay of a 2-input XOR or XNOR gate. As always, n is the total number of bits in a data word, so n/2 is the number of bits of the module. The module’s delay is:

$$\text{delay} = a \log_2\left(\frac{n}{2}\right) + b\cdot\left(\sum_{i=1}^{\log_2(n/2)} (2^i+1)\right) + d_{XOR} \tag{7}$$

4.5 Prefix incrementer and its delay

Another implementation for the incrementer based on the prefix carry-lookahead generator follows. It uses more gates and interconnections but has a smaller delay since each gate load is reduced.

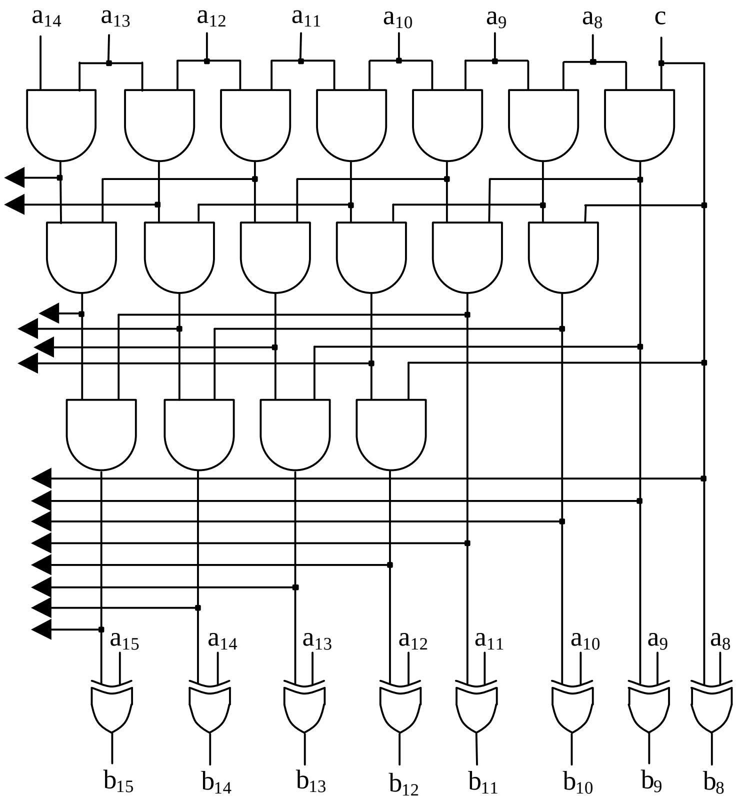


Figure 11. Conditional incrementer. Prefix implementation.

The delay is

$$\text{delay}_{\text{prefix}} = \log_2(n/2)\ (\ a + 2\ b) + d_{XOR} \tag{8}$$

4.6 Complete conversion circuit

Because of the comparison instructions, it is not enough to add the c-bit to the upper half of the redundant result in order to obtain the result in conventional representation. In case a comparison (SLT) is performed, the result is either a 1 or a 0 depending on the sign of the converted value.

Figure 12 shows the logic necessary for the comparisons. Note that this logic is the same logic that should be present as the last stage of a conventional ALU. It is not present in our proposed ALU. It has to be present instead as part of the conversion circuit together with the conditional incrementer.

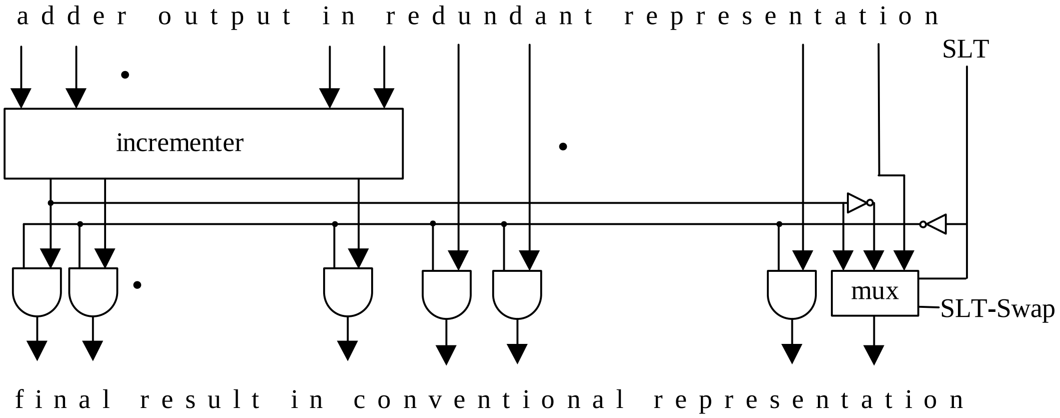

adder output in redundant representation
SLT
incrementer
mux
SLT-Swap
final result in conventional representation

Figure 12: Extra logic necessary for the comparisons.

4.7 Overflow detection

The overflow can be detected in the usual manner, i.e. checking the sign and the carry from a_{n-2} to a_{n-1} for signed overflow, carry out for unsigned overflow. However, the redundant representation considered can store higher values, up to

$$\text{maximum value} = \left(\sum_{i=0}^{n-1} 2^i\right) + 2^{n/2} = 2^n - 1 + 2^{n/2} \tag{9}$$

Therefore, overflow will be detected in two steps.

1. While adding, overflow is detected only if the resulting value is greater than the maximum value in Equation (7).
2. While converting, overflow is detected for those values between 2^n-1 and the maximum value in Equation (7). In this case, a change in the value's sign (most significant bit) implies signed overflow.

4.8 Radix-$2^{n/3}$ and -$2^{n/4}$ conversion circuit

In this section, the increase in complexity of the conversion circuit when going to lower radices is discussed.

Increasing the amount of redundancy (i.e. using lower radices) reduces the Execution Stage time. It should be done if after using radix-$2^{n/2}$ the critical path is still there. The problem, however, is that the conversion time increases rapidly so that the approach developed in this thesis is estimated to be good without major modification for radices $2^{n/2}$, $2^{n/3}$ and $2^{n/4}$ only.

The problems found when increasing the redundancy are

1. the incrementer needs more inputs,
2. the incrementer requires an OR gate, and

3. the incrementer’s total number of gates grows too rapidly

The factors above cause the conversion for radices $2^{n/3}$ and $2^{n/4}$ to need two more levels of gates for n=128, while for radices below those the complexity of the operation approaches that of the regular 2 n-bit operand sum.

V. PERFORMANCE IMPROVEMENTS

5.1 Introduction

In this chapter, some techniques and modules that complement and enhance the basic pipeline implementation described in the first chapter are discussed.

These techniques belong to four different groups

1. Stall rate reduction. The first approach deals with avoiding the stall necessary when a subtraction is data dependent on the preceding instruction and the dependency involves the subtrahend. The second one is more generic. It consists in the redundant addition providing a conventional instead of redundant result, which can be done in most cases, as will be shown in this Chapter.

2. Memory cycle time reduction. It was stated in the first chapter that an increase in memory access time was needed because of the address conversion time. However, depending on the actual implementation of the memory subsystem, this overhead can be avoided.

3. Radix-$2^{n/2}$ carry save representation for instruction address adders. The techniques used for the ALU adder can also be used for the two other adders in the MIPS R2000 pipeline.

5.2 Subtraction-caused stall-rate reduction

One source of stalls is subtraction of a redundant subtrahend. The stall is needed when subtracting the result of a previous addition or another subtraction. It can be avoided by rearranging the sequence of additions performed to an equivalent series, either making use of the compiler or dynamically. For example, the sequence ‘a-(b+c)’ can be changed to ‘(a-b)-c’ to avoid the stall. Similarly, ‘a-(b-c)’, can be changed to ‘(a-b)+c’.

The transformations above work fine when storing the result of the two-operand intermediate addition in a fourth (temporary) register. However, problems can arise when using one of the operands to store the result if there are no spare registers. Consider the assignments ‘r = a-(b+c)’, that can be changed to ‘r = (a-b)-c’. And consider, on the other hand, ‘c = a-(b+c)’, which cannot be changed to ‘c = (a-b)-c’ without introducing an additional intermediate register to store (a-b).

Other cases cannot be solved in the same way. For example, consider the next code fragment, where the fact that two assignments are made prevent the stall to be avoided.

```
int a, b;
[...]
```

```
a = a + a;  /* multiply ‘a’ by two and store back in ‘a’ */
b = b - a;  /* b= b-(original a +original a) */
```

In the example above the stall can only be avoided by moving a previous or posterior instruction in between the addition and the subtraction.

5.3 Fast conversion to conventional representation

The idea here is to not use the $\log_2(n/2)$ 2-input ANDs plus XOR gate delay incrementer to get the conventional representation, but, when possible, get the result with one or two levels of two-input gates instead.

The fast conversion logic is shown in Figures 1 through 3.

In Figure 13, advantage is taken of the fact that a redundant representation of a value is identical to its conventional representation when its c-bit is zero.

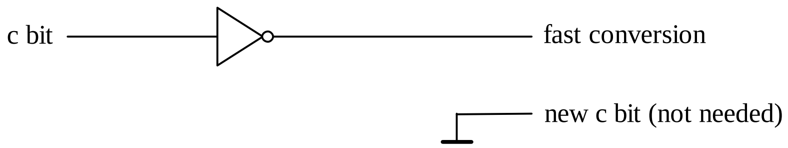


Figure 13. Fast conversion logic. It avoids 50% of the subtraction-caused stalls for random inputs. The conversion can be done whenever ‘fast conversion’=1.

When the c-bit is one but the data bit with its same weight is zero, both bits can be swapped to be able to discard the c-bit and obtain the conventional representation directly. The logic circuit in Figure 14 expands the one in Figure 1 while taking the latter fact into account.

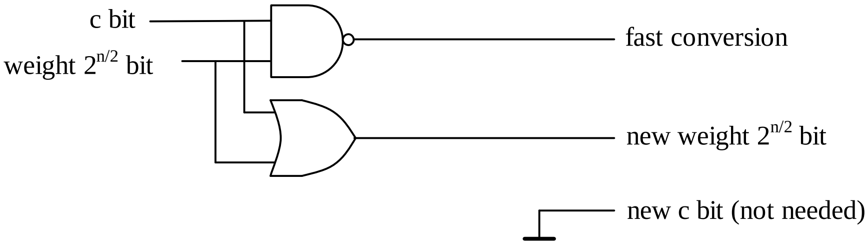


Figure 14. Fast conversion logic. It avoids 75% of the subtraction-caused stalls for random inputs. Note that the ‘new c’ and ‘new weight...’ bits are only correct if ‘fast conversion’=1.

Figure 15 expands the reasoning to one more bit. In general, what is done is checking the increment carry-propagation chain. A variable time addition is then performed, where the time employed –either none or one more cycle- is dependent on the data.

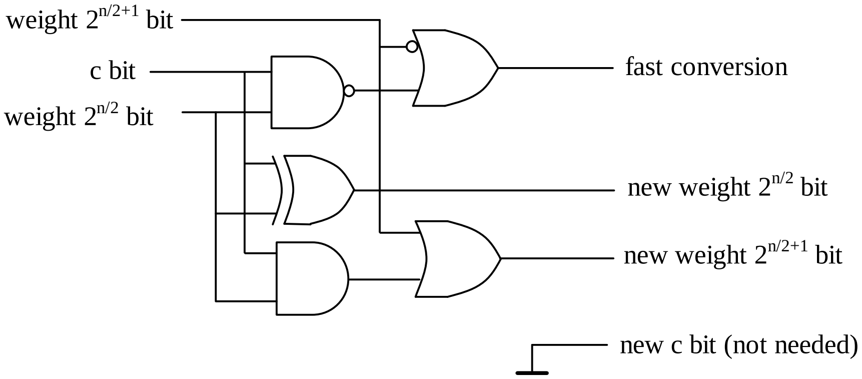


Figure 15. Fast conversion logic. It avoids 87.5% of the subtraction-caused stalls for random inputs. Note that the ‘new...’ bits are only correct if ‘fast conversion’=1.

Note that although a fast conversion can be applicable sometimes, it adds to the cycle time when it is used. If the needed extra time is not available a prediction scheme (“c-bit prediction”) could be preferred, even having a lower conversion rate. The c-bit prediction can also be combined with the fast-conversion scheme.

5.4 c-bit prediction

Since doing a fast conversion can still take too long, a better solution is to try to predict the c-bit in parallel with the addition. Then, some time before the result is ready it will be known whether the result is conventional (c-bit=0) or redundant (c-bit=1), or its representation is still not known. In the latter case, the result is safely assumed to be redundant.

The idea is that when there is a carry kill in the n/2-1 position (the $2^{n/2-1}$ weight bits of both operands are 0) the c bit will be 0 regardless of the values of the other bits. This situation (the $2^{n/2-1}$ weight bits of both operands are 0) will occur one fourth of the time for uniformly distributed random inputs but more than that if small numbers (those under $2^{(n-1)/2}$) are frequent.

By looking at the values of more bits, the ratio of non-redundant results (c-bit predicted as zero) tends to 50%. However, when the c-bit is predicted to be one, a fast conversion can still be used to improve the rate of conventional values.

5.5 Conversion time reduction for memory stage

The incrementer before the data-cache address input can be avoided by making use of a redundantly addressable memory system as shown in Figure 16. This substitution can potentially reduce the memory stage cycle time, as compared to the redundant pipeline proposed in Chapter 1.

Blocks useful for the proposed configuration are the ‘redundantly addressable memory’ (memory with a 1 position shifter after the decoder) and the ‘sum addressable memory’ (a simplified version for radix-$2^{n/2}$ is enough). See also [3] for further details.

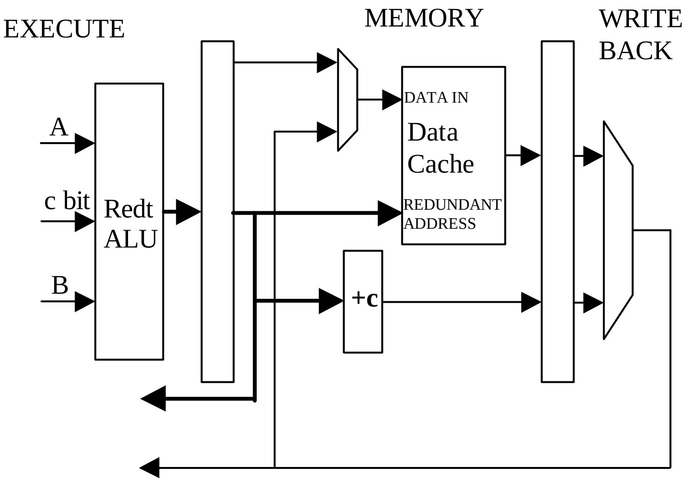


Figure 16
This is like (a fragment of) Fig 5 in Chapter 1, except that a redundantly addressable memory system is used. The only difference with Fig 5 is that the redundant ALU output value is used instead of the converted one to address the data memory in the memory stage.

Figure 17 shows the modified data memory subsystem for a MIPS R10000 processor. In this case, the latency is increased by only one level of Full Adders, which are necessary to implement the comparator of the redundantly addressable CAM (Contents addressable memory), as is shown in Fig. 18.

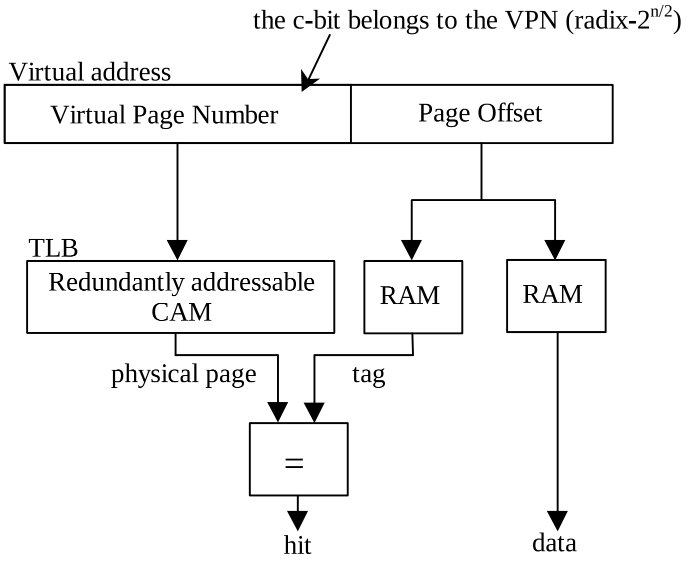


Figure 17: Memory hierarchy. A 1-way cache and TLB system similar to that described in [1] but modified for redundancy is used.

While using radix-$2^{n/2}$ the c-bit is likely to belong to the virtual page number, not to the page offset. Therefore, only the TLB has to deal with the redundancy. However, if using lower radices it could be necessary to change the regular RAMs in the figure to redundantly-addressable ones. These are discussed in section “5.7 Redundantly addressable memory”.

5.6 Redundantly addressable CAM (contents addressable memory)

This kind of memory is able to handle a redundant value as address. Conversion to conventional is not done. Instead, the built-in comparators need an extra level of adders to perform its function. Note that the overhead is constant time, independent of n (even the time of the whole comparator being not), and independent of the amount of redundancy used.

This comparator is a simplified version of the one described in [6].

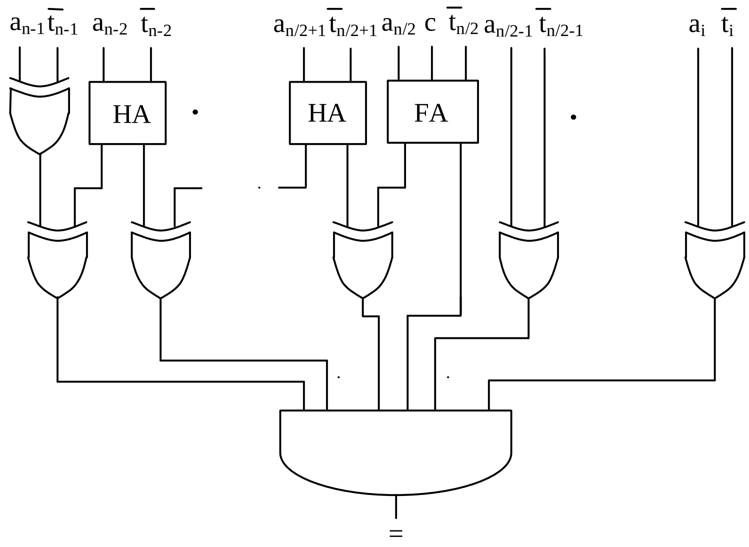


Figure 18: Redundant comparator of a redundantly addressable CAM.

Figure 19 shows the implementation at the transistor level of a CAM comparator cell. It is based on the regular CAM comparator described in [8]. The ‘bit’ signal is the input (a_i in Fig. 18). ‘tag’ is the stored tag bit (t_i in Fig.18). In the implementation shown a half-adder is implemented, the XOR gate is substituted by a XNOR, and the AND gate by a NOR one.

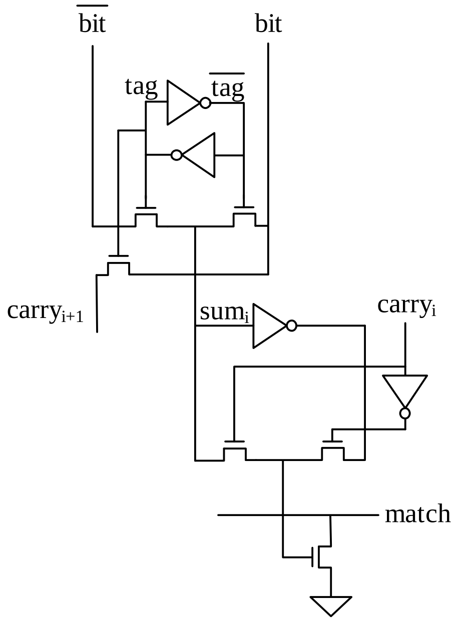


Figure 19: Possible transistor implementation for the Redundant-CAM-cell. Only the Half-Adder based cell is shown. The write lines are not shown.

5.7 Redundantly addressable memory

This kind of memory is shown in Fig. 20. It is able to handle a radix-$2^{n/2}$ redundant value as address. Conversion to conventional is not done. Instead, a shift of the same amount as the weight of the c bit is performed on the decoded value. If two or more c-bits are present (radix-$2^{n/3}$, -$2^{n/4}$...) more shifters are needed.

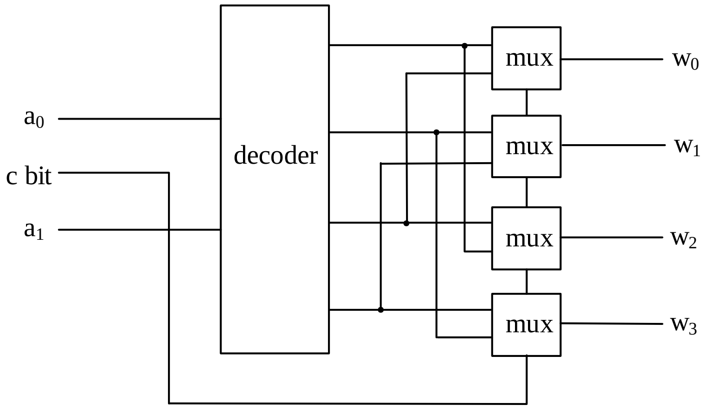


Figure 20: Redundant decoder for n=2. A shift (or, as in this example, a rotation) of the same amount as the weight of the c bit is performed on the decoded conventional value.

5.8 Instruction address adders

In addition to the ALU adder, there are two more adders in other stages of the pipeline, which could be in the critical path when expanding the word width. The PC adder, that determines the next instruction address when executing in sequence, and the branch adder, that determines it when taking a branch, can also take advantage of a redundant representation to speed up.

The redundant representation used for these adders is not exactly the same as used for the ALU adder. The extra c-bit is not an internal not-propagated carry coming from the previous addition but is, instead, a prediction of what the c-bit will be in the next addition. It would only be necessary to use the usual redundant representation when a ‘jump to address in register x’ is used, with register ‘x’ just out of the ALU. The conventional representation would be used in this case to load the PC, via stall, in order to avoid inconsistencies.

Figure 21 shows the address adders for a regular MIPS implementation. This is a detail of the pipeline shown in Figure 4.

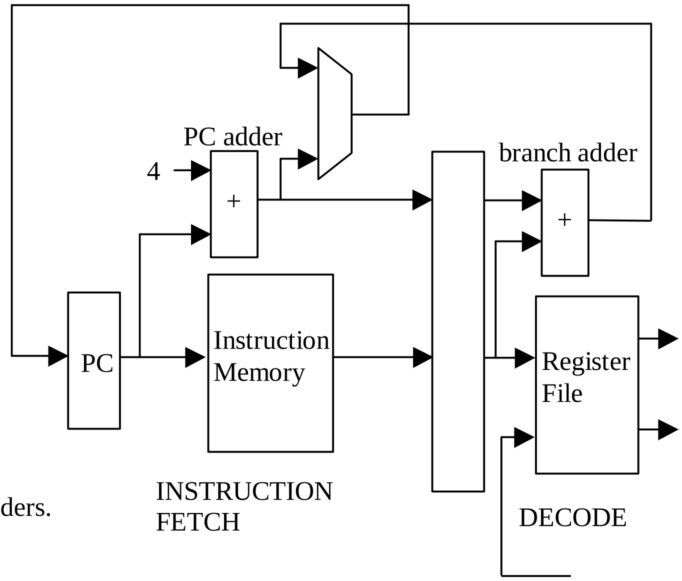
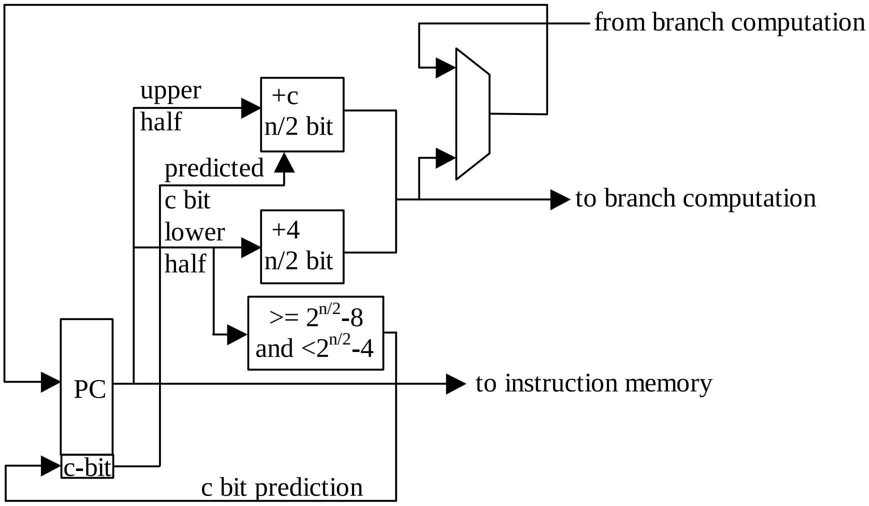


Figure 21:
PC and branch adders.

5.9 Program Counter adder

The PC adder must deliver a sequentially increased PC value every clock cycle. To do that in a redundant representation would imply the memory to be able to be addressed in redundant representation. If this is not convenient or possible (it is, in principle, possible, as is shown in sections “5.7 Redundantly addressable memory” and the following), another approach is to predict the carry to the higher half of the address word. This is not difficult since the amount to be added to the PC is constant every cycle, and thus known.

Figure 22
PC adder,
redundant.



5.10 Branch adder

The branch adder can be moved to later stages. This provides more time (i.e. two cycles instead of one) for the addition to be performed, but on the other hand makes the branches more expensive (i.e. the latency will be increased by one cycle).

A better solution is to predict the carry into the most significant half word. Note that it should be zero most of the time. It only will be one when crossing the boundaries between two blocks of size $2^{n/2}$ words. Also, the cost in extra hardware and control is low since the recovery from a mispredicted carry is not different from the recovery from a mispredicted branch.

5.11 Radix-$2^{n/3}$ and radix-$2^{n/4}$ carry-save pipeline

The same concept developed for radix-$2^{n/2}$ carry-save can be generalized to more redundancy, i.e. lowest radices and consequently more c-bits. The increased redundancy allows for a reduced addition time. However, it also implies that the conversion time is longer. The conversion times for diverse amounts of redundancy are discussed in the 4th chapter, “Conversion Circuit”.

5.12 Shifter

In the first chapter, it was assumed that a shift didn’t ever take more time than a redundant addition. However, shifts take logarithmic time, as additions do, and as addition time is reduced making use of redundancy, the likeliness of the cycle time being too short to perform a shift, increases.

A good solution to this problem is to pipeline the shift. The small shifts are performed in the execution stage and the large ones are finished in the memory stage. This will work well assuming the former are more likely to happen. The 2nd shift stage then is placed together with the conversion.

VI. CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

In this work it has been shown that it is possible to use redundancy in microprocessors in order to reduce addition times. The addition time can be cut by 10% for radix-$2^{n/2}$ carry-save redundancy and 20% for radix-$2^{n/4}$. If the pipeline's critical path is in the execution stage the cycle time can be cut by about 5% or 10%, respectively.

The advantage of using redundancy versus pipelining the adders is that no stalls are necessary for two consecutive data-dependent additions. However, other problems do exist. One of these problems, which is examined very closely in the main text, is called representation-dependency.

Some different types of representation dependencies are discussed for a particular architecture (MIPS). Representation dependencies appear when a particular series of two instructions presents a data dependency. The series of instructions affected are:

1. A set-less-than instruction followed by an addition or subtraction

2. An addition or subtraction followed by a subtraction, but only if the data-dependency affects the subtrahend.

One technique is necessary in order to keep the previous list that short. This is the SLT operand swap. Other techniques are developed and discussed that reduce the impact of the representation dependencies. These include: Rescheduling of representation-dependent instructions, c-bit prediction and fast conversion of redundant values, all of which can be combined for maximum improvement.

In the worst case, representation-dependencies lead to a stall of the pipeline. The stall rate due to representation dependencies can be kept below 1% by making use of the techniques above.

Another problem is the necessity of converting the redundant value in order to address the memory system. Some solutions are discussed, that include different types of redundantly-addressable memories that are developed in this work. As an example, a solution is presented for a modified MIPS R10000 pipeline's data-cache and virtual-memory subsystem that presents very little overhead.

Logic-level implementations, as well as delay estimations, are presented for the redundant pipeline, the redundant ALU, the redundant memory, the redundant CAM, the redundant comparator, the incrementer circuit and the complete conversion circuit. A transistor-level implementation of the comparator is shown that would be used as a cache cell.

Similar techniques to those used for the execution stage are presented for the additions in the pipeline other than the in the ALU: The branch adder and the program-counter incrementer. Implementations are showed for the redundant branch adder and the redundant program counter incrementer.

6.2 Future work

- Lower radices, down to radix-2 carry-save

  To achieve further reduction of the addition time, higher levels of redundancy are to be used. However, conversion time increases along with the redundancy. Using radix-2, an addition of two conventional operands takes no time but its conversion is a full conventional addition.

- Signed digit redundancy

  As has been shown, one of the problems of radix-2^m carry-save redundancy is performing a 2's complement. Other redundant representations that completely avoid this problem should be considered.

- Modified instruction set

  Using the redundant addition can be advantageous in some circumstances and disadvantageous in other. An instruction set that included separate redundant-addition and conventional-addition instructions would allow for complete control for the compiler or programmer, thus allowing an optimum performance improvement.

- Influence on superscalar machines

  No attempt has been done to determine which new problems could arise if using the proposed modifications on a superscalar machine.

- Influence of flag-based instruction sets

  A RISC architecture that uses no flags has been considered in this work. No attempt has been done to examine other problems that could arise when applying the exposed ideas to some popular CISC architectures.

- Floating point unit

  Similar techniques to those used for the integer unit should be applied to the floating-point unit.

EFFECT OF HIGH-RADIX CARRY-SAVE REDUNDANCY IN ALU ON PROCESSOR CYCLE TIME, ARCHITECTURE, AND IMPLEMENTATION

REFERENCES

[1] Kenneth C. Yeager “The MIPS R10000 Superscalar Microprocessor”, IEEE Micro, April 1996, pp28-40.

[2] Sunil Mirapuri et al. “The MIPS R4000 Processor”, IEEE Micro, April 1992, pp10-22.

[3] William L. Lynch et al. “Low Load Latency through Sum-Addressed Memory (SAM)”, 14th Symposium on Computer Arithmetic Proceedings, 1998, pp369-379.

[4] John L. Hennessy and David A. Patterson. “Computer Architecture: A Quantitative Approach”, Morgan Kauffman Publishers, Inc., San Francisco, California, second edition, 1996.

[5] David R. Lutz and D. N. Jayasimha “The Half Adder Form and Early Branch Condition Resolution”, 13th Symposium on Computer Arithmetic Proceedings, Asilomar, California, July 1997, pp266-273.

[6] Jordi Cortadella and Jose M. Llaberia. “Evaluation of A+B=K conditions without carry propagation”, IEEE Transactions on Computers, 41(11): 1484-1488, November 1992.

[7] M.D. Erzegovac and T. Lang. “Digital Arithmetic” (Draft), September 1998

[8] Weste and Eshraghian. “Principles of CMOS VLSI Design”, 2nd ed., Addison Wesley, 1993.

[9] Samuel Naffziger. “ A Sub-Nanosecond 0.5µm 64b Adder Design”, 1996 IEEE International Solid-State Circuits Conference, Digest of Technical Papers, ISSCC, IEEE, 1996, pp.362-3

APPENDIX

EVALUATION OF CYCLE TIME REDUCTION FOR VARIOUS ADDER IMPLEMENTATIONS

A.1 Evaluation of cycle time reduction for various adder implementations

In this Appendix, the time required for redundant addition and conversion is evaluated. Some of the results presented here are used in the 3rd chapter, “Performance Evaluation”.

Two adders are considered, and a variation on the first one.

1. Multilevel carry look-ahead adder
2. Conditional sum adder

The assumptions are that the addition takes only a fraction of the original cycle time, and that the other fraction remains unchanged.

The speed up, or performance improvement, is computed using the formula,

$$\text{improvement} = 1 - \frac{T_{NE}\,(1+\text{stall}_N)}{T_{OE}\,(1+\text{stall}_O)}$$

Which assumes that the execution stage time, (the old, T_{OE}, and the new, T_{NE}) limit both the old and new frequencies.

The stall rates are not considered in this Appendix. We are concerned only with the

$$\text{improvement}_{\text{without stalls}} = 1 - \frac{T_{NE}}{T_{OE}}$$

A.2 Multilevel carry look-ahead adder

The expression for the delay of a regular multilevel carry look-ahead adder is (from [7])

$$T_{\text{max-regCLA}} = t_{p,g} + (\log_2 n + 1)\, t_{clg} + t_s$$

and for a prefix CLA,

$$T_{\text{max-prefixCLA}} = t_{p,g} + (\log_2 n)\, t_{clg} + t_s$$

Where $t_{p,g}$ is the time it takes to generate the initial propagation and generation signals, n is the number of bits in a word, t_{clg} is the time it takes to generate the intermediate propagation and generation signals, and t_s is the time to perform the sum given all the other signals. We are using groups of two bits. It is the most usual choice in VLSI design.

However, as the loads for each pg-generation module are many and (for the regular CLA) vary greatly, should also be taken into account separately. Assuming that each additional load increases the delay in 20% that of a gate delay, we have the following.

$$T_{max\text{-}regCLA} = t_{p,g} + (\log_2 n + 1)\, t_{clg} + t_s + 20\% L_n$$

where the number of loads, L_n, found in the gates in the critical path, if unbuffered, is

$$L_n = 2 + \sum_{i=1}^{\log_2 n} (2^{i-1}+1)$$

but, if buffered, we can take the value of L_n as being

$$L_n = 2 \log_2 n + 10 \log_5 n \text{ [equivalent loads]}$$

Where the first addend accounts for two loads per level except for the one with the largest load, and the second accounts for the level with largest load, where a tree of buffers is placed. The buffers are supposed to have a delay expression of 1+20%L equivalent gate delays.

The corresponding expression for the prefix CLA, is

$$T_{max\text{-}prefixCLA} = t_{p,g} + \log_2 n\, (t_{clg} + 0.4) + t_s$$

0.4 being the extra delay due to a load of two.

On the other hand, when delivering the result in radix $2^{n/2}$ carry save representation the time required is that of an n-1 bits wide adder. Thus,

$$T_{max\text{-}radix2^{n/2}regCLA} = t_{p,g} + (\log_2 n)\, t_{clg} + t_s + 20\% L_{n/2}$$

$$T_{max\text{-}radix2^{n/2}prefixCLA} = t_{p,g} + (\log_2 n - 1)\, (t_{clg} + 0.4) + t_s$$

In Tables A1 and A2 there are the evaluations of the times and improvements for several word widths, assuming the values for $t_{p,g}$, t_{clg}, and t_s stated in the tables.

α is the ratio $T_{OVERHEAD}/T_{ADDER}$, where T_{ADDER} and $T_{OVERHEAD}$ are the two components of the execution stage time, $T_E=T_{ADDER}+T_{OVERHEAD}$. ($\alpha$ is the overhead, the addition time taken as unity).

Table A1: Evaluation of increase of performance for radix $2^{n/2}$ multilevel REGULAR carry-lookahead adders with BUFFERING

						α=1		
						cycle time	cycle time	improvement, %
n	lg n	L_n	$L_{n/2}$	T_{max}CLA	$T_{max}R2^{n/2}$	original	improved	(w/o stalls)
8	3	19	13	18	14	36	32	11
16	4	25	19	22	18	43	39	8.7
32	5	32	25	25	22	51	47	**7.4**
64	6	38	32	29	25	58	54	**6.5**
128	7	44	38	33	29	66	62	**5.7**
256	8	50	44	37	33	73	69	**5.1**
512	9	57	50	40	37	81	77	4.7
1024	10	63	57	44	40	88	84	4.3

$t_{p,g}$	t_{clg}	t_s	
2	2.5	2 levels of gates	times in '# of gate levels'

Table A1: Evaluation of increase of performance for radix $2^{n/2}$
(continued) multilevel REGULAR carry-lookahead adders with BUFFERING

α=0.33			α=3		
cycle time original	cycle time improved	improvement, % (w/o stalls)	cycle time original	cycle time improved	improvement, % (w/o stalls)
24	20	16	71	67	5.3
29	25	13	86	82	4.4
34	30	**11**	101	97	**3.7**
39	35	**9.7**	116	113	**3.2**
44	40	**8.6**	131	128	**2.9**
49	45	**7.7**	146	143	**2.6**
54	50	7.0	161	158	2.3
59	55	6.4	176	173	2.1

Table A2: Evaluation of increase of performance for radix $2^{n/2}$ multilevel PREFIX carry-lookahead adders

					α=1		
n	lg n	lg n -1	TmaxCLA	TmaxR2n/2	cycle time original	cycle time improved	improvement, % (w/o stalls)
8	3	2	13	10	25	23	11
16	4	3	16	13	31	28	9.3
32	5	4	19	16	37	34	**7.8**
64	6	5	22	19	43	40	**6.8**
128	7	6	25	22	49	46	**6.0**
256	8	7	28	25	54	52	**5.3**
512	9	8	31	28	60	57	4.8
1024	10	9	34	31	66	63	4.4
	$t_{p,g}$	t_{clg}	t_s				
	2	2.5	2 levels of gates			times in '# of gate levels'	

Table A2: Evaluation of increase of performance for radix $2^{n/2}$
(continued) multilevel PREFIX carry-lookahead adders

α=0.33			α=3		
cycle time original	cycle time improved	improvement, %	cycle time original	cycle time improved	improvement, % (w/o stalls)
17	14	17	51	48	5.7
21	18	14	62	60	4.6
25	22	**12**	74	71	**3.9**
28	26	**10**	86	83	**3.4**
32	29	**9.0**	97	94	**3.0**
36	33	**8.0**	109	106	**2.7**
40	37	7.2	120	118	2.4
44	41	6.6	132	129	2.2

A.3 Conditional sum adder

The expressions are (from [7])

$$T_{\text{max-CondSA}} = t_{\text{add}} + (\log_2 n)\, t_{\text{mux}}$$

$$T_{\text{max-radix}2^{n/2}\text{CondSA}} = t_{\text{add}} + (\log_2 n - 1)\, t_{\text{mux}}$$

Being t_{add} the time of a 2-bit adder and t_{mux} the time of a multiplexer. We are assuming groups of 2 bits.

See Table A3 for the evaluation table for various values of n. We assume t_{add}=4, and t_{mux}=2 levels of gates.

Buffering times of the carry signals are not taken into account. They are not negligible but can be avoided using look-ahead (see [9]).

Table A3: Evaluation of increase of performance for radix $2^{n/2}$ conditional sum adders					α=1		
n	lg n	lg n -1	T_{max}CondS	T_{max}rad$2^{n/2}$	cycle time, original	cycle time, improved	improvement, % (w/o stalls)
8	3	2	10	8	20	18	10
16	4	3	12	10	24	22	8.3
32	5	4	14	12	28	26	**7.1**
64	6	5	16	14	32	30	**6.2**
128	7	6	18	16	36	34	**5.5**
256	8	7	20	18	40	38	**5.0**
512	9	8	22	20	44	42	4.5
1024	10	9	24	22	48	46	4.1
	t_{add}	t_{mux}					
	4	2	levels of gates			(times in '# of gate levels')	

Table A3, (ct'd)					
α=0.33			α=3		
cycle time original	cycle time improved	improvement, % (w/o stalls)	cycle time original	cycle time improved	improvement, % (w/o stalls)
13	11	15	40	38	5.0
16	14	12	48	46	4.2
19	17	**11**	56	54	**3.6**
21	19	**9.4**	64	62	**3.1**
24	22	**8.4**	72	70	**2.8**
27	25	**7.5**	80	78	**2.5**
29	27	6.8	88	86	2.3
32	30	6.3	96	94	2.1

In addition to the increment in performance, note that roughly one third of the adder area is saved when using redundancy.

A.4 Evaluation of increase in performance for radix 2^m carry look-ahead adders

The performance is evaluated using, as before, the assumption that the overhead time (originally a fraction (α/1+α) of the cycle time) remains unchanged. The addition time (originally 1/1+α) changes from an n-bit wide addition to an m-bit wide one, being m equal to n divided by one plus the number of redundancy bits.

Thus, the improvement is

$$\text{improvement}_{\text{without stalls}} = 1 - \frac{\alpha\, T_{\text{n-bitAddition}} + T_{\text{m-bitAddition}}}{(1+\alpha)\, T_{\text{n-bitAddition}}}$$

The expression for the time required for an m-bit carry look-ahead addition is

$$T_{\text{max-CLA}} = t_{p,g} + (\log_2 m - 1)\, t_{clg} + t_s + 20\% L_m$$

Table A4:
Evaluation of increase in performance for 2^m carry-lookahead adders
LOADING considered

	α=1				
	#redundancy bits				
n	0	1	2	3	7
8	0	11	17	21	32
16	0	8.7	14	17	26
32	0	**7.4**	**12**	**15**	**23**
64	0	**6.5**	**10**	**13**	**19**
128	0	**5.7**	**9.1**	**11**	**17**
256	0	**5.1**	**8.1**	**10**	**15**
512	0	4.7	7.4	9.3	14
1024	0	4.3	6.8	8.5	12
$t_{p,g}$	t_{clg}	t_s			
2	2.5	2	levels of gates		

Table A4 (ct'd)		improvement, %, w/o stalls					
α=.33				α=3			
#redundancy bits				**#redundancy bits**			
1	**2**	**3**	**7**	**1**	**2**	**3**	**7**
16	25	32	48	5.3	8.4	11	16
13	21	26	39	4.4	6.9	8.7	13
11	**18**	**22**	**33**	**3.7**	**5.9**	**7.4**	**11**
9.7	**15**	**20**	**29**	**3.2**	**5.1**	**6.5**	**9.7**
8.6	**14**	**17**	**26**	**2.9**	**4.5**	**5.7**	**8.6**
7.7	**12**	**15**	**23**	**2.6**	**4.1**	**5.1**	**7.7**
7.0	11	14	21	2.3	3.7	4.7	7.0
6.4	10	13	19	2.1	3.4	4.3	6.4

The numbers above assume that the performance is only limited by the execution stage, what is less likely to remain true as the level of redundancy increases.

A.5 Delay of prefix incrementer

In order to evaluate the overhead introduced by the conversion to conventional representation, the tables below show the proportion of time used by such conversion with respect to the cycle time allowed by the improved adder. The tables are only for radix-$2^{n/2}$ carry save representation.

Table A5: Incrementer time, with respect to the prefixCLA cycle time							
		α=1		α =0.33		α =3	
n	incr. Time	cycle time, improved	%	cycle time, improved	%	cycle time, improved	%
8	4.3	23	19	14	31	48	9
16	5.7	28	20	18	32	60	10
32	7.1	34	**21**	22	**33**	71	**10**
64	8.5	40	**21**	26	**33**	83	**10**
128	9.9	46	**22**	29	**34**	94	**10**
256	11.3	52	**22**	33	**34**	106	**11**
512	12.7	57	22	37	34	118	11
1024	14.1	63	22	41	34	129	11

Table A6:
Incrementer time, with respect to the Conditional Sum Adder cycle time

		α =1		α =0.33		α =3	
		cycle time,		cycle time,		cycle time,	
n	incr. Time	improved	%	improved	%	improved	%
8	4.3	18	24	11	38	38	11
16	5.7	22	26	14	41	46	12
32	7.1	26	**27**	17	**43**	54	**13**
64	8.5	30	**28**	19	**44**	62	**14**
128	9.9	34	**29**	22	**45**	70	**14**
256	11.3	38	**30**	25	**46**	78	**14**
512	12.7	42	30	27	47	86	15
1024	14.1	46	31	30	47	94	15

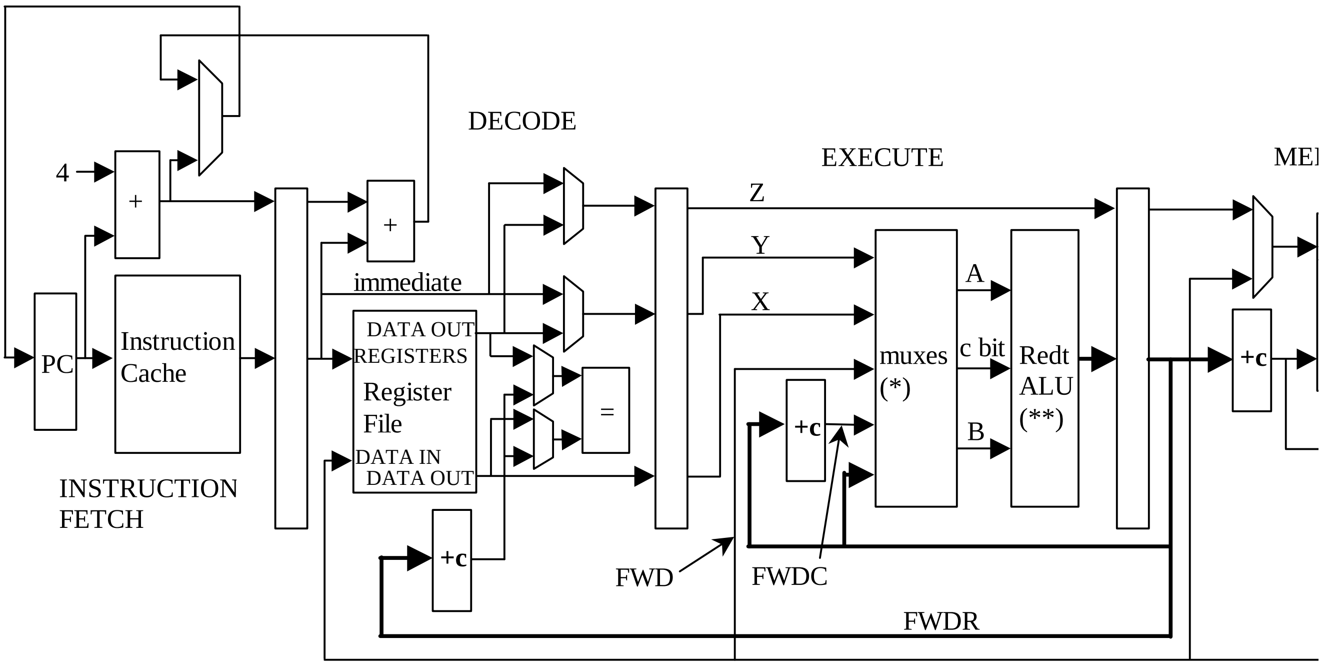


Figure 5
Data-path of modified implementation. The conditional incrementer ('+c' block) is shown three times in order to clarify it affects. Thick lines represent radix 2 $^{n/2}$ buses, which are n+1 bits wide. Trapezoids are multiplexers. Control lines are
(*) These multiplexers are shown in detail in Figure 7.
(**) The Redundant ALU is shown in detail in Figure 6.

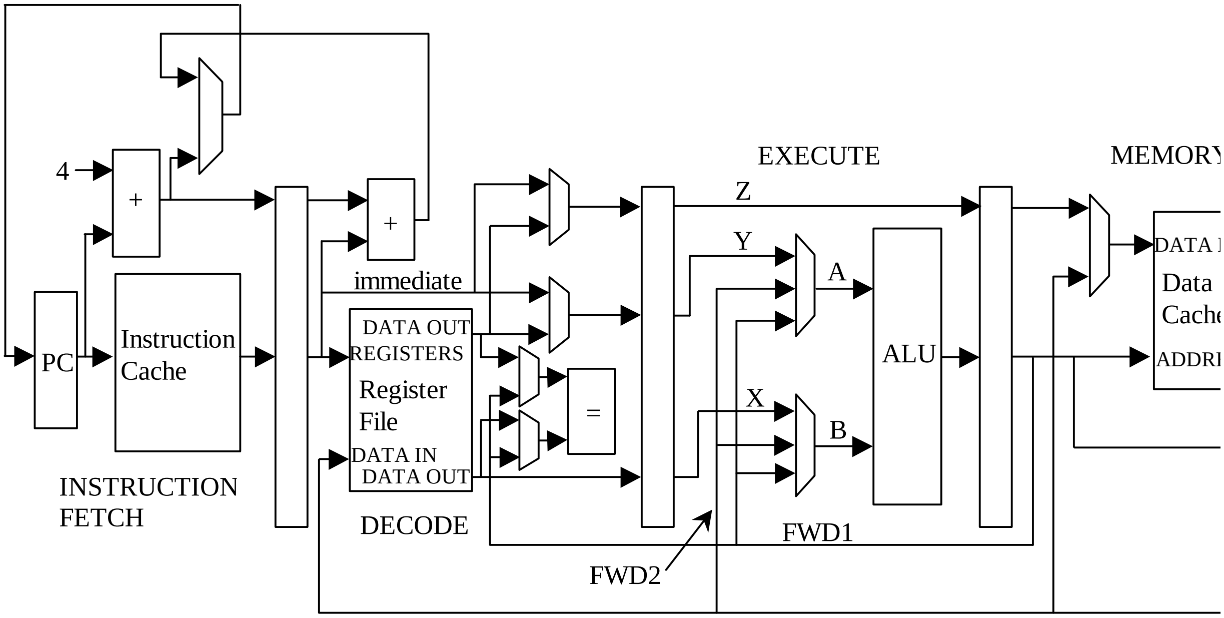


Figure 4
MIPS architecture data-path implementation to be modified. Control lines not shown.

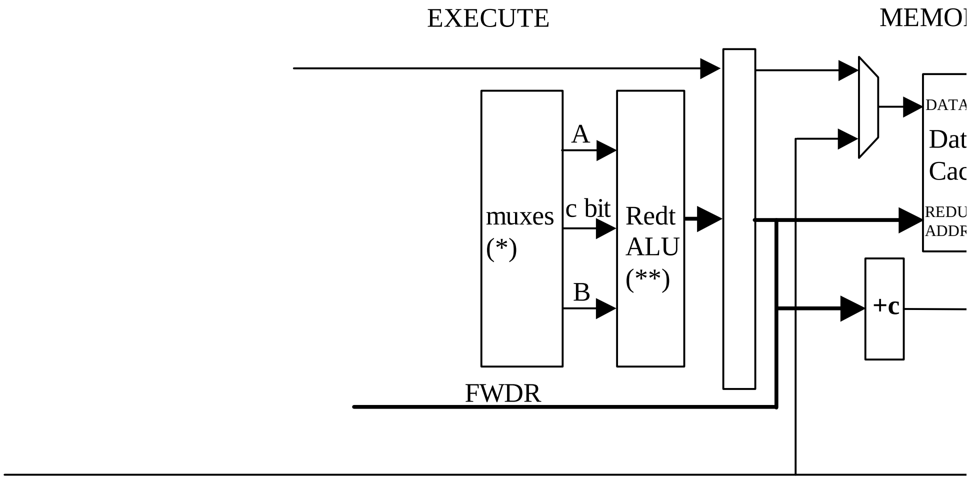


Figure 7
This is like Fig 3 except that a redundantly addressable memory is used. The only difference with Fig 3 is that the red output value is used instead of the converted one to address the data memory in the memory stage.