Programmer Reference Manual
SIMD Enhanced MIPS Instructions

Ellen Burger (ID#: 015263571)
Yung Nguyen (ID#: 015599829)

CECS 341 MW 8:00AM - 10:15AM
Due Date: December 3, 2018
R. W. Allison

Table of Contents

I. Purpose

A. Abstract

The objective of this manual is to outline the implementation of various SIMD enhancements to MIPS architecture. Existing MIPS framework such as register set, addressing modes, and data types are explained as an introduction, as well as instructions that are used to build SIMD instructions. The new commands are based on eight-byte vectors with two-byte wide elements with two additional similar types. In MIPS, they are split between two to four registers. The SIMD enhancements are split into three sections: summaries detailing each new instruction, source code to implement to them, and logs showing the successfully calculated result. It also details how the existing MIPS datapath would need to be amended to properly implement the vector instructions for efficiency. The manual concludes with final thoughts and comments on the process it took to create this.

B. Enhanced Instruction Set

The SIMD enhancements are intended to be used by software developers in MIPS assembly language. Due to MIPS's basic functionality, powerful tools such as vectors can't be directly implemented without ad hoc solutions. These upgrades intends to remedy that.

The new instructions provide developers with tools to save time and increase efficiency. Vectors allow large quantities of data to be processed at once and the SIMD lets them be used with needing to write out lengthy, complex code; it is already done for the user. These can be implemented into existing algorithms seamlessly. For example, a process needing to total values from large quantities of vectors can be simplified with the use of the vec_addsu command.

In addition, instructions like vec_encry and vec_sortup allow vectors to be manipulated to a greater degree. This is to create a wide field of functionality covering needs from security to database organization. The SIMD enhancements outlined in this manual are intended to be succinct and effective for all possible vector needs.

II. Instruction Set Architecture

A. Machine Register Set

REGISTER NAME, NUMBER, USE, CALL CONVENTION

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$v1	2-3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	Yes

MIPS has thirty-two registers available to the user for performing routines and operations. They're denoted as $n where n is the numbered value (0-31) or $xn where xn is the name.

- Register **$zero** is the constant 0 and remains unchanged.
- Registers **$at, $k0, and $k1** are reserved for the assembler to use.
- Registers **$v0-$v1** are used to return results from functions and subroutines.
- Registers **$a0-$a3** are used as arguments in subroutines.
- Registers **$t0-$t7** are temporary registers used for values that don't need to be preserved across calls to other instructions.
- Registers **$s0-$s7** are saved registers that are used for values that should be saved across calls.
- Register **$gp** is the global pointer which points to the middle of the heap. The heap is a 64K memory block containing constants.
- Register **$sp** is the stack pointer. It always points to the top of the stack, ready to push or pop from it.
- Register **$fp** is the frame pointer. It contains the stack pointer's value at the start of a routine to prevent it from being lost.
- Register **$ra** is the return address stored when the jal command is used. It's often used to return to the location jumped from using jr.

B. Data Types

1. Byte

4-bit value used to store integers with no decimal places. It can also be used to create an array.

```
byte1: .byte 621
byte2: .byte 12,13,14,15
```

2. Word

32-bit value used to store most types of data. It can be positive or negative. It can also be used to create an array.

```
word1: .word 15
word2: .word -20
array: .word 0,1,2,3,4
```

3. Half-Word

16-bit value used to store half of a word.

```
halfword: .halfword 0x1234
```

4. Float

32-bit value used to store a numerical value with one decimal value.

```
float1: .float 4.1
float2: .float 12.5
```

5. Double

32-bit value used to store integers with multiple decimal values.

```
double1: .double 4.13
double2: .double 64.128
```

C. Addressing Modes

1. Register

Description: Instructions where all operands accessed by an operation are within registers.

Example: sub $s0, $t0, $t1

31 26 25 21 20 16 15 11 10 6 5 0

0	0x8	0x9	0x10	0x8	0

```
$t0       = 0x00000045
$t1       = 0x00000015
$t0 - $t1 = 0x00000030
$s0       = 0x00000030
```

Other examples: add, or, sll

2. Register Indirect

Description: Instructions where the effective address accessed by the operation is within a register.

Example: jr $ra

31 26	25 21	20 16	15 11	10 6	5 0
0	0x1F	0	0	0	0x8

```
$ra = 0x00000040
PC  = 0x00000040
```

3. Base Offset

Description: Instructions where the effective address is the sum of the 16-bit immediate value and a register.

Example: lw $s1, 4 ($a0)

31 26	25 21	20 16	15 0
0x23	0x4	0x11	0x4

```
[$a0]          = 0x00000040
offset         = 0x00000004
[$a0] + offset = 0x00000044
$s1            =[0x00000044]
```

Other examples: sw

4. PC-Relative

Description: Instructions where the effective address is the current PC value + 4 offset by the immediate value.

Example: beq $zero, $a2, Label

NOTE: where the target is two instructions from the next instruction.

31 26	25 21	20 16	15 0
0x4	0x0	0x6	0x10

```
$zero =       0x00000000
$a2 =         0x00000000
PC =          0x0000000C
PC + 4 =      0x00000010
PC + 2 * 4 = 0x00000018
```

Other examples: bne

5. Immediate

I-type instructions where an operation is conducted using an immediate 16-bit operand.

Example: addi $v0, $t4, 16

31 26	25 21	20 16	15 0
0x8	0xC	0x2	0x10

```
$t4        = 0x00000040
imm        = 0x00000010
$t4 + imm = 0x00000050
$v0        = 0x00000050
```

Other examples: li, slti

D. Instruction Set (with user examples) and Binary Instruction Formats

NOTE: Values will be represented in hex

1. Triple Operand Instructions

add $rd, $rs, $rt	Adds rs and rt and stores sum into rd
and $rd, $rs, $rt	Bitwise ands rs to rt and stores result into rd
or $rd, $rs, $rt	Bitwise ors rs to rt and stores result into rd
sll $rd, $rs, h	Shifts the value in rs left by the amount of bits specified in h. Zeroes are shifted in. Result stores into rd
srl $rd, $rs, h	Shifts the value in rs right by the amount of bits specified in h. Zeroes are shifted in. Result stores into rd
slt $rd, $rs, $rt	Checks if rs is less than rt. Stores 1 into rd if so; otherwise stores 0
sub $rd, $rs, $rt	Subtracts rt from rs and stores difference into rd

1.1 add

31 – 26	25 – 21	20 – 16	15 – 11	10 – 06	05 – 0
0x0	rs	rt	rd	0x0	0x20
Opcode	rs	rt	rd	shamt	funct

Syntax: add $rd, $rs, $rt

Operation: R[$rd] ← R[$rs] + R[$rt]

Description: Adds two registers ($rs and $rtt) and stores the result in a register ($rd)

Example:

`Code: add $t2,$t0,$t1`

31 – 26	25 – 21	20 – 16	15 – 11	10 – 06	05 – 0
0x0	rs	rt	rd	0x0	0x20
Opcode	rs	rt	rd	shamt	funct

```
$t0 = 0x00000020
$t1 = 0x00000040
⇒ $t2 = 0x00000060
```

1.2 and

31 – 26	25 – 21	20 – 16	15 – 11	10 – 06	05 – 0
0x0	rs	rt	rd	0x0	0x24
Opcode	rs	rt	rd	shamt	funct

Syntax: and $rd, $rs, $rt

Operation: R[$rd] ← R[$rs] & R[$rt]

Description: Bitwise ands two registers ($rs and $rt) and stores the result in a register

Example:

`Code: and $t2,$t0,$t1`

```
$t0 = 0x00000056
$t1 = 0x00000057
⇒ $t2 = 0x00000056
```

1.3 or

31 – 26	25 – 21	20 – 16	15 – 11	10 – 06	05 – 0
0x0	rs	rt	rd	0x0	0x25
Opcode	rs	rt	rd	shamt	funct

Syntax: or $rd, $rs, $rt

Operation: R[$rd] ← R[$rs] | R[$rt]

Description: Bitwise ors two registers ($rs and $rt) and stores the result in a register

Example:

`Code: or $t2,$t0,$t1`

```
$t0 = 0x00000056
$t1 = 0x00000057
⇒ $t2 = 0x00000057
```

1.4 sll

31 – 26	25 – 21	20 – 16	15 – 11	10 – 06	05 – 0
0x0	0x0	rt	rd	sa	0x0
Opcode	rs	rt	rd	shamt	funct

Syntax: sll $rd, $rt, shamt

Operation: R[$rd] ← R[$rt] << shamt

Description: Shifts a register value (in $rt) left by the shift amount (shmt) listed in the instruction and stores the result in a register ($rd). Zeroes are shifted in.

Example:

Code: `sll $t1,$t0,4`

`$t0 = 0x00000056`

⇒ `$t1 = 0x00000560`

1.5 srl

31 – 26	25 – 21	20 – 16	15 – 11	10 – 06	05 – 0
0x0	0x0	rt	rd	sa	0x2
Opcode	rs	rt	rd	shamt	funct

Syntax: srl $rd, $rt, shamt

Operation: R[$rd] ← R[$rt] >> shamt

Description: Shifts a register value (in $rt) right by the shift amount (shmt) listed in the instruction and stores the result in a register ($rd). Zeroes are shifted in.

Example:

Code: `srl $t1,$t0,4`

`$t0 = 0x00000056`

⇒ `$t1 = 0x00000005`

1.6 sub

31 – 26	25 – 21	20 – 16	15 – 11	10 – 06	05 – 0
0x0	rs	rt	rd	0x0	0x22
Opcode	rs	rt	rd	shamt	funct

Syntax: sub $rd, $rs, $rt

Operation: R[$rd] ← R[$rs] - R[$rt]

Description: Subtracts rt from rs and stores difference into rd

Example:

Code: sub $t2,$t0,$t1

```
$t0 = 0x00000056
$t1 = 0x00000050
⇒ $t2 = 0x00000006
```

2. Double Operand Instructions

div $rs, $rt	Divide rs by rt and store the quotient into $LO and the remainder into $HI
mult $rs, $rt	Multiplies rs by rt and store the result into $LO

2.1 div

31 – 26	25 – 21	20 – 16	15 – 11	10 – 06	05 – 0
0x0	rs	rt	rd	0x0	0x1a
Opcode	rs	rt	rd	shamt	funct

Syntax: div $rs, $rt

Operation: LO ← R[$rs] / R[$rt]
HI ← R[$rs] % R[$rt]

Description: Divide rs by rt and store the quotient into $LO and the remainder into $HI

Example:

Code: div $t0,$t1

```
$t0 = 0x00000056

$t1 = 0x00000056

⇒   HI = 0x00000000

    LO = 0x00000001
```

2.2 mult

31 – 26	25 – 21	20 – 16	15 – 11	10 – 06	05 – 0
0x0	rs	rt	rd	0x0	0x18
Opcode	rs	rt	rd	shamt	funct

Syntax: mult $rs, $rt

Operation: {HI, LO} ← R[$rs] * R[$rt]

Description: Multiplies rs by rt. The resulted low order word is stored in $LO and the resulted high order word is stored in $HI.

Example:

Code: mult $t0,$t1

```
$t0 = 0x12345678

$t1 = 0x00000100

⇒   HI = 0x00000012

    LO = 0x34567800
```

3. Single Operand Instructions

mfhi $rd	Stores the contents of $HI into rd
mflo $rd	Stores the contents of $LO into rd

3.1 mfhi

31 – 26	25 – 21	20 – 16	15 – 11	10 – 06	05 – 0
0x0	0x0	0x0	rd	0x0	0x10
Opcode	rs	rt	rd	shamt	funct

Syntax: mfhi $rd

Operation: R[$rd] ← HI

Description: Stores the contents of $HI into rd

Example:

```
Code:     Mult $t0, $t1
          mfhi $t2
$t0 = 0x12345678
$t1 = 0x00000100
⇒   HI = 0x00000012
    LO = 0x34567800
⇒   $t2 = 0x00000012
```

3.2 mflo

31 – 26	25 – 21	20 – 16	15 – 11	10 – 06	05 – 0
0x0	0x0	0x0	rd	0x0	0x12
Opcode	rs	rt	rd	shamt	funct

Syntax: mflo $rd

Operation: R[$rd] ← LO

Description: Stores the contents of $LO into rd

Example:

```
Code:     Mult $t0, $t1
          mflo $t2
$t0 = 0x12345678
$t1 = 0x00000100
⇒   HI = 0x00000012
    LO = 0x34567800
⇒   $t2 = 0x34567800
```

4. Conditional Branches

beq $rs, $rt, offset	If rs = rt, branch to the pc location specified by offset. pc + offset << 2
bne $rs, $rt, offset	If rs != rt, branch to the pc location specified by offset. pc + offset << 2

4.1 beq

31 – 26	25 – 21	20 – 16	15 – 0
0x4	rs	rt	offset
opcode	rs	rt	immediate

Syntax: beq $rs, $rt, offset

Operation: if(R[$rs] = R[$rt])
PC ← PC + 4 + $SignExt_{18b}$({offset, 00})

Description: If rs = rt, branch to the pc location specified by offset. pc + offset << 2

Example:

Code:

```
Loop:
subi $t0,$t0,1
beq $t0, $t1,Loop
...
```

```
$t0 = 0x00000006
$t1 = 0x00000005
```

⇒ `$t0 = 0x00000004`

4.2 bne

31 – 26	25 – 21	20 – 16	15 – 0
0x4	rs	rt	offset
opcode	rs	rt	immediate

Syntax: bne $rs, $rt, offset

Operation: if(R[$rs] != R[$rt])
PC ← PC + 4 + $SignExt_{18b}$({offset, 00})

Description: If rs != rt, branch to the pc location specified by offset. pc + offset << 2

Example:

Code:

```
Loop:
addi $t0,$t0,1
bne $t0, $t1,Loop
...
```

```
$t0 = 0x00000000

$t1 = 0x00000005
```

⇒ `$t0 = 0x00000005`

5. Unconditional Jump and Subroutine Call/Return Instructions

j target	Jump to the address specified by the immediate value, target. PC = target << 2
jal target	Jump to the address specified by the immediate value, target. PC = target << 2. The current PC value is stored into $ra
jr $rs	Jump to the address stored in rs. PC = rs.
syscall	Exits a routine using the value in v0

5.1 j

31 – 26	25 – 0
0x2	Target address
Opcode	address

Syntax: j target

Operation: PC ← {(PC + 4)[31:28], target, 00}

Description:Jump to the address specified by the immediate value, target. PC = target << 2

Example:

```
Code:     J JDest
          addi $t0,$t0,2
          JDest:
          Subi $t0,$t0,2
          ...
$t0 = 0x000000020
⇒   $t0 = 0x0000001E
```

5.2 jal target

31	26 25 0
0x3	Target address
Opcode	address

Syntax: jal target

Operation: R[31] ← PC + 8
PC ← {(PC + 4)[31:28], target, 00}

Description:Jump to the address specified by the immediate value, target. PC = target << 2. The current PC value is stored into $ra

Example:

```
Code:                jal  JDest
          exit:      ori  $v0, $zero, 10
                     syscall
                     JDest:
                     subi $t0,$t0,2
                     jr $ra                    #return
```

$t0 = 0x000000020

⇒ $t0 = 0x0000001E

5.3 jr

31 – 26	25 – 21	20 – 16	15 – 11	10 – 06	05 – 0
0x0	rs	0x0	0x0	0x0	0x08
Opcode	rs	rt	rd	shamt	funct

Syntax: jr $rs

Operation: PC ← R[$rs]

Description: Jump to the address stored in rs. PC = rs

Example:

```
Code:                jal  JDest
          exit:      ori  $v0, $zero, 10
                     syscall
                     JDest:
                     subi $t0,$t0,2
                     jr $ra                    #return
```

$t0 = 0x0000020

⇒ $t0 = 0x0000001E

5.4 syscall

31 26	25 21	20 16	15 11	10 06	05 0
0x0	xx	xx	xx	xx	0x12
Opcode	rs	rt	rd	shamt	funct

Syntax: syscall

Operation: System Call

Description: Exits a routine using the value in v0

Example:

```
Code:     exit:     ori  $v0, $zero, 10
                    syscall
```

6. Immediate Operand Instructions

addi $rt, $rs, imm	Adds $rs with a sign-extended immediate value and stores into $rt
li $rt, c	Loads the 16-bit immediate value c and stores into rt. li is a pseudo instruction that calls the addiu instruction to implement
slti $rt, $rs, imm	Checks if rs is less than the immediate value. Stores 1 into rt if so; otherwise stores 0

6.1 addi

31 – 26	25 – 21	20 – 16	15 – 00
0x8	rs	rt	imm
Opcode	rs	rt	imm

addi $rt, $rs, imm

Operation: R[$rt] ← R[$rs] - imm

Description: Subtracts an immediate value from rs and stores difference into rt

Example:

Code: addi $t2,$t0,0x00000001

$t0 = 0x0000008

⇒ $t2 = 0x00000009

6.2 slti

31 – 26	25 – 21	20 – 16	15 – 00
0xA	rs	rt	imm
Opcode	rs	rt	imm

Syntax: slti $rt, $rs, imm

Operation: If R[$s] < imm, R[$rt] = 1. If not, R[$rt] = 0

Description: Compares rs to an immediate value. If it is less than, rt is set to one; otherwise, it is set to 0

Example:

Code: slti $t2,$t0,0x00000016

$t0 = 0x0000008

⇒ $t2 = 0x00000001

7. Single Instruction Multiple Data (SIMD) Instructions

Each SIMD implementation will involve the use of **vectors.** They will be represented by the MIPS 32-bit registers. Two registers will be combined to form one 64-bit “vector register” for the SIMD instructions. Each element within these vector registers will be 8-byte each.

Each SIMD vector registers specifies “even-addressed” registers. So, a SIMD instruction specifying vector registers $t0, $t2, $t6 as operands, would have its element carried out in the 64-bit registers of $t0:$t1, $t2:$t3, and $t6:$t7.

Example: vec_mule d,a,b → vec_mule $t0, $t2,$t6

0	1	2	3	4	5	6	7
$t2	$t2	$t2	$t2	$t3	$t3	$t3	$t3

Figure: vector a

SIMD implementation also uses vectors featuring 16-byte wide elements, four total elements in 64-bytes.

Example: vec_mule d,a,b → vec_mule $t0, $t2,$t6

0	1	2	3
$t0	$t0	$t1	$t1

Figure: vector d

Additionally, SIMD specifies a 128-byte wide vector with 16-byte wide elements. Eight elements total and is contained over four registers.

Example: vec_encry d,a,b → vec_encry $t0, $t4,$t6

0	1	2	3	4	5	6	7
$t0	$t0	$t1	$t1	$t2	$t2	$t3	$t3

Figure: vector d

Baseline SIMD Enhancements

7.1 Vector Add Saturated (unsigned)

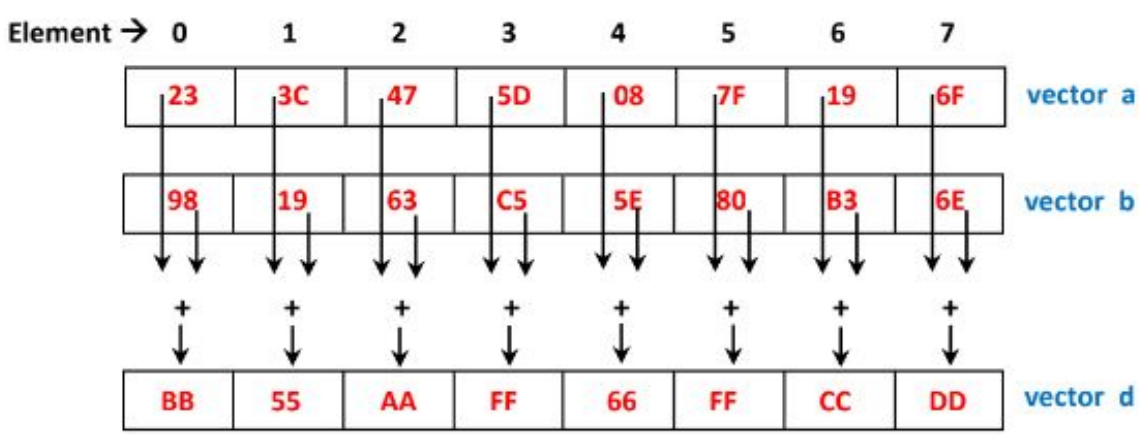
31 – 28	27 – 20	19 – 15	14 – 10	9 – 5	4 – 0
0111	00000000	rs	rt	00000	rd
Constant	Opcode	rs	rt	ru	rd

Syntax: vec_addsu d,a ,b

Vectors Used: a, b, d

Description: Each element (8 byte) in vector a and the corresponding element (8 byte) in vector b is added together, resulting in the unsigned-integer that will be placed in the corresponding element in vector d

Representation:



7.2 Vector Multiply and Add

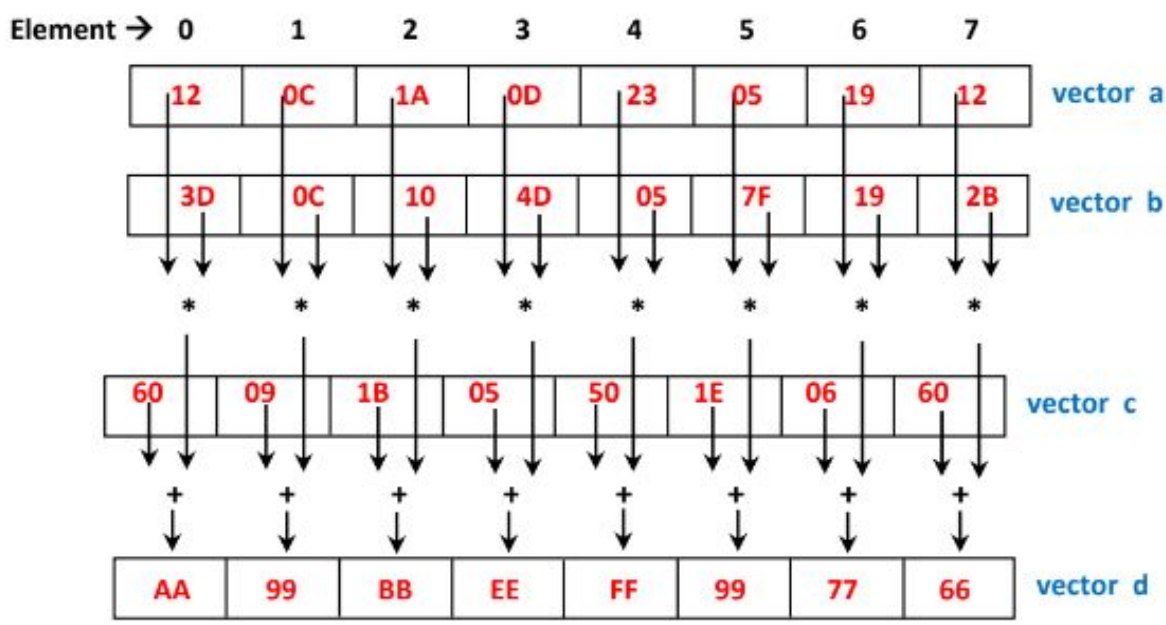
31 – 28	27 – 20	19 – 15	14 – 10	9 – 5	4 – 0
0111	00000001	rs	rt	ru	rd
Constant	Opcode	rs	rt	ru	rd

Syntax: vec_madd d, a, b, c

Vectors Used: a, b, c, d

Description: Multiply each element in vector a by the corresponding element in vector b. The product will then be added with the corresponding element in vector c. The sum will be stored in the corresponding element in vector d.

Representation:



7.3 Vector Multiply Even Integer

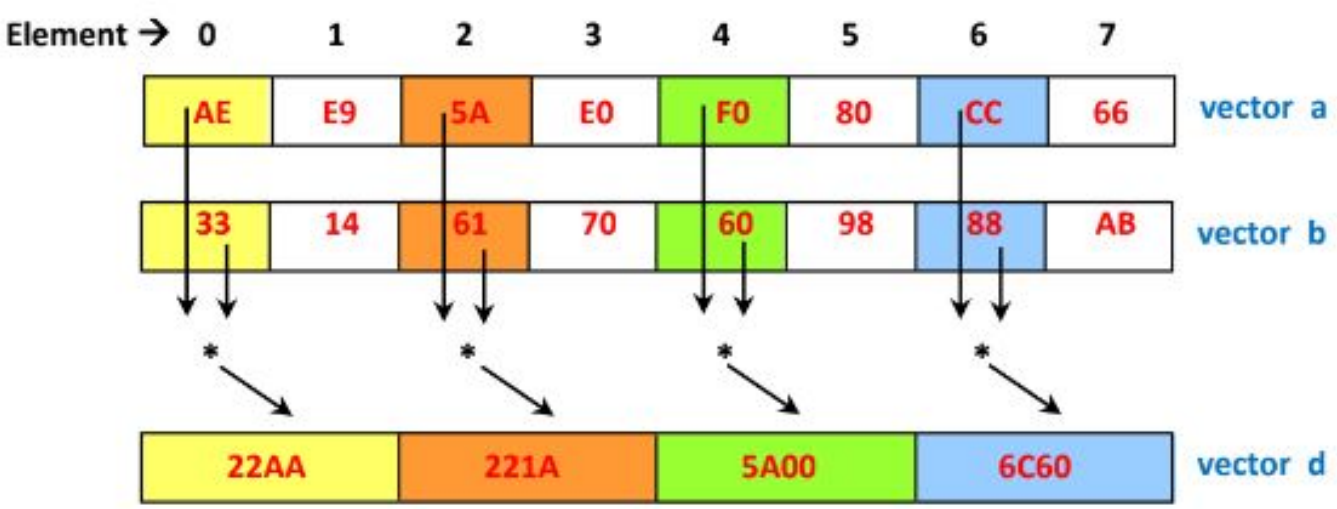
31 – 28	27 – 20	19 – 15	14 – 10	9 – 5	4 – 0
0111	00000010	rs	rt	00000	rd
Constant	Opcode	rs	rt	ru	rd

Syntax: vec_mule d, a, b

Vectors Used: a, b, d

Description: Each high half-width part of each element in vector a is multiplied with each corresponding high half-width part of each element in vector b. The product (full 16-bit) is stored in the vector d

Representation:



7.4 Vector Multiply Odd Integer

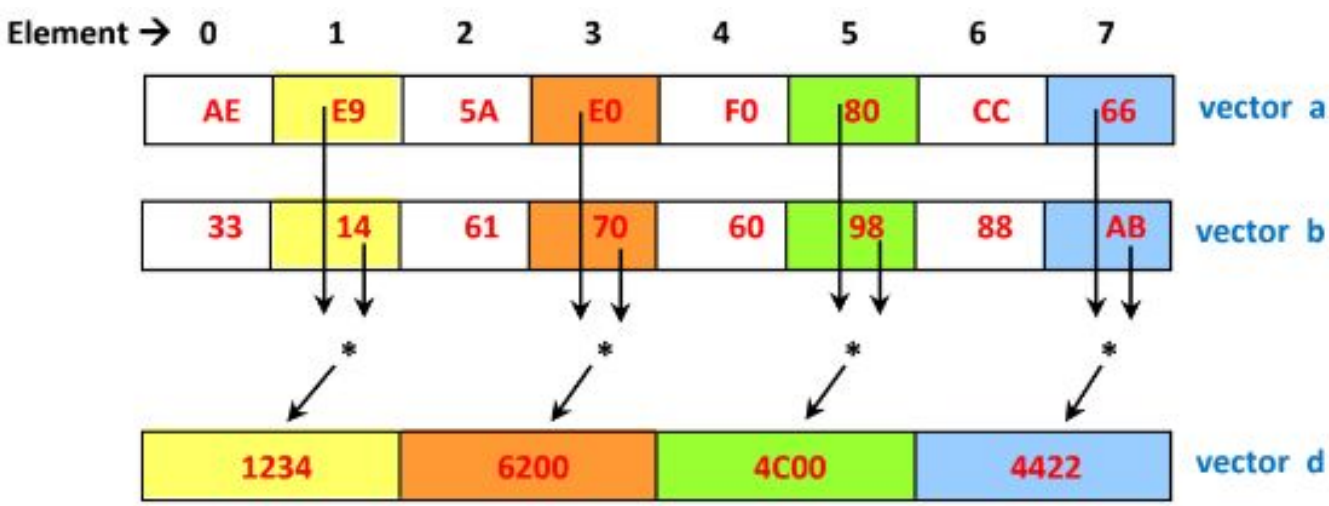
31 – 28	27 – 20	19 – 15	14 – 10	9 – 5	4 – 0
0111	00000011	rs	rt	00000	rd
Constant	Opcode	rs	rt	ru	rd

Syntax: vec_mulo d, a, b

Vectors Used: a, b, d

Description: Each low half-width part of each element in vector a is multiplied with each corresponding low half-width part of each element in vector b. The product (full 16-bit) is stored in the vector d

Representation:



7.5 Vector Multiply Sum Saturated

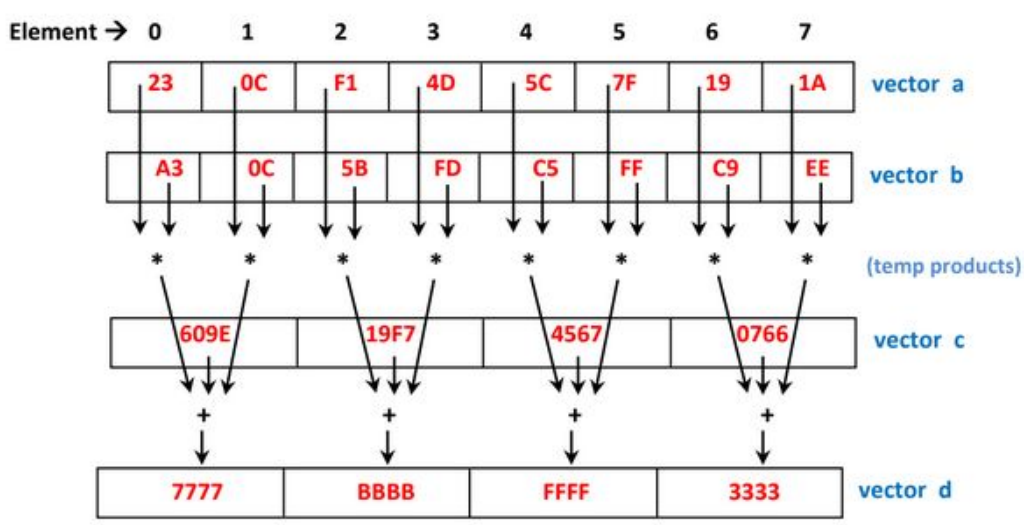
31 – 28	27 – 20	19 – 15	14 – 10	9 – 5	4 – 0
0111	00000100	rs	rt	ru	rd
Constant	Opcode	rs	rt	ru	rd

Syntax: vec_msums d, a, b, c

Vectors Used: a, b,c, d

Description: Each consecutive 8-bit elements in vector a are multiplied to the corresponding consecutive 8-bit elements in vector b, both of which corresponds to the 16-bit elements in vector c. The 16-bit “temp” products are then added with the corresponding 16-bit elements in vector c. Their sums are stored in corresponding element in vector d.

Representation



7.6 Vector Splat

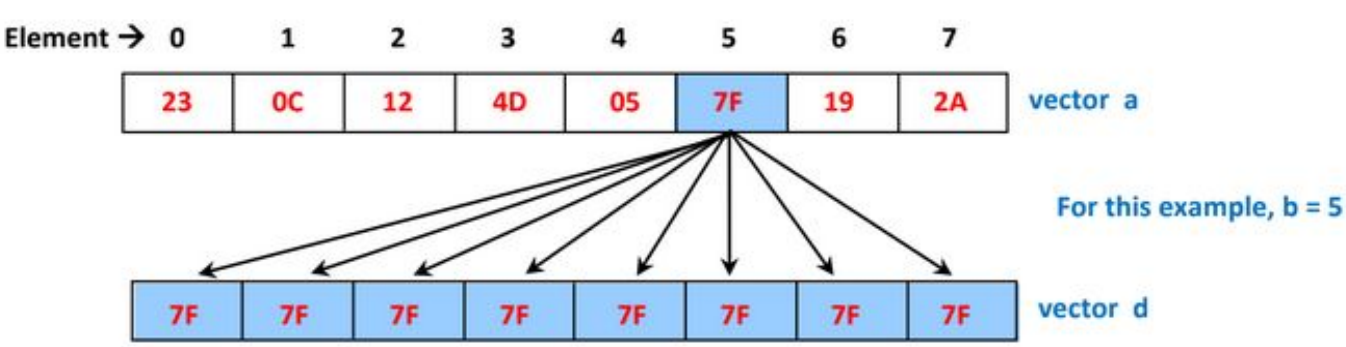
31 – 28	27 – 20	19 – 15	14 – 10	9 – 5	4 – 0
0111	00000101	rs	rt	00000	rd
Constant	Opcode	rs	rt	ru	rd

Syntax: vec_splat d, a, b

Vectors Used: a, d

Description: An element from vector a will be copied to every single element in vector d, using b as the specifier of which element from vector to choose from.

Representation:



7.7 Vector Merge Low

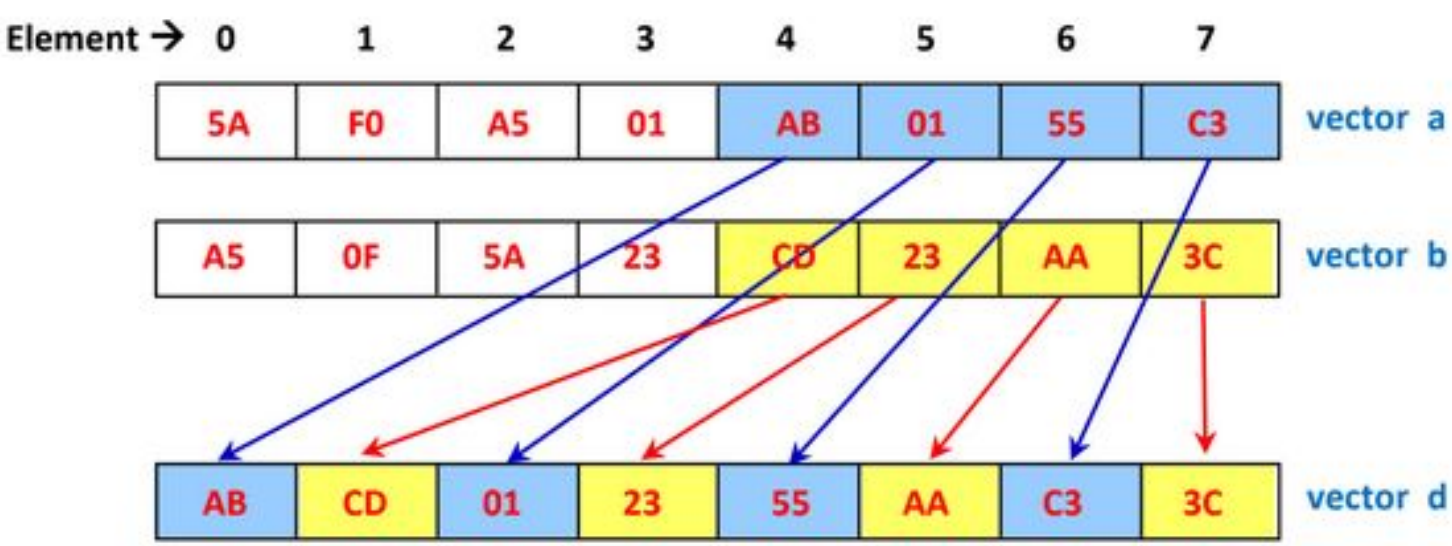
31 – 28	27 – 20	19 – 15	14 – 10	9 – 5	4 – 0
0111	00000110	rs	rt	00000	rd
Constant	Opcode	rs	rt	ru	rd

Syntax: vec_mergel d, a, b

Vectors Used: a, b, d

Description: The low elements in vector a are stored in the even elements in vector d. The low elements in vector b are stored in the odd elements in vector d.

Representation:



7.8 Vector Merge High

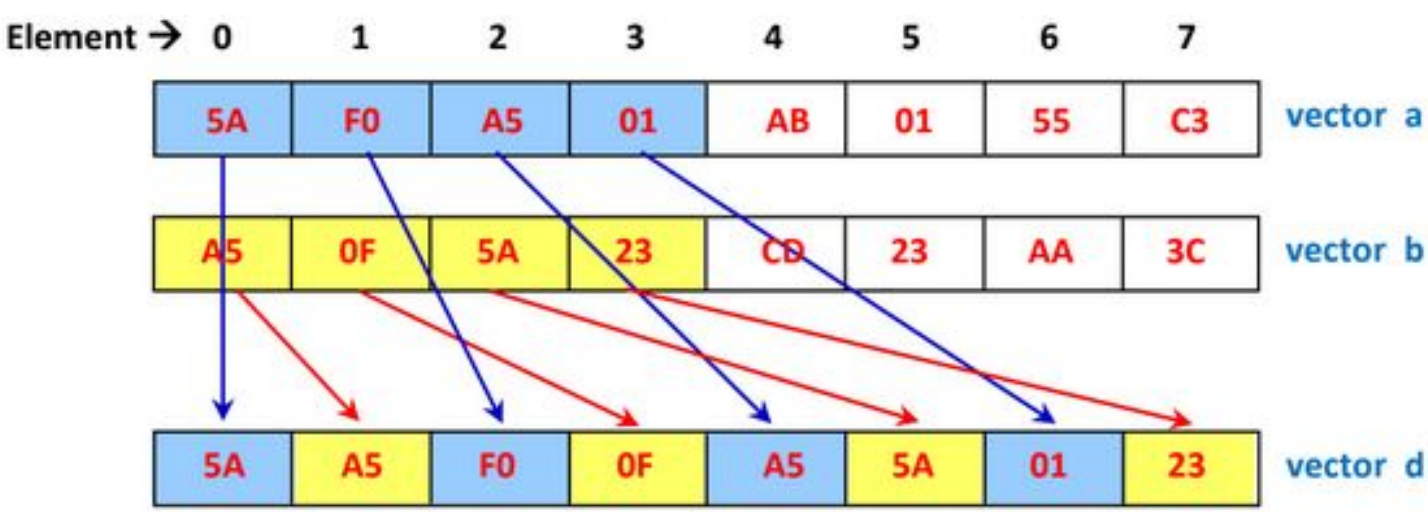
31 – 28	27 – 20	19 – 15	14 – 10	9 – 5	4 – 0
0111	00000111	rs	rt	00000	rd
Constant	Opcode	rs	rt	ru	rd

Syntax: vec_mergeh d, a, b

Vectors Used: a, b, d

Description: Similar to **Number 7**, the high (instead of low) elements in vector a are stored in the even elements in vector d. The high (instead of low) elements in vector b are stored in the odd elements in vector d.

Representation:



7.9 Vector Pack

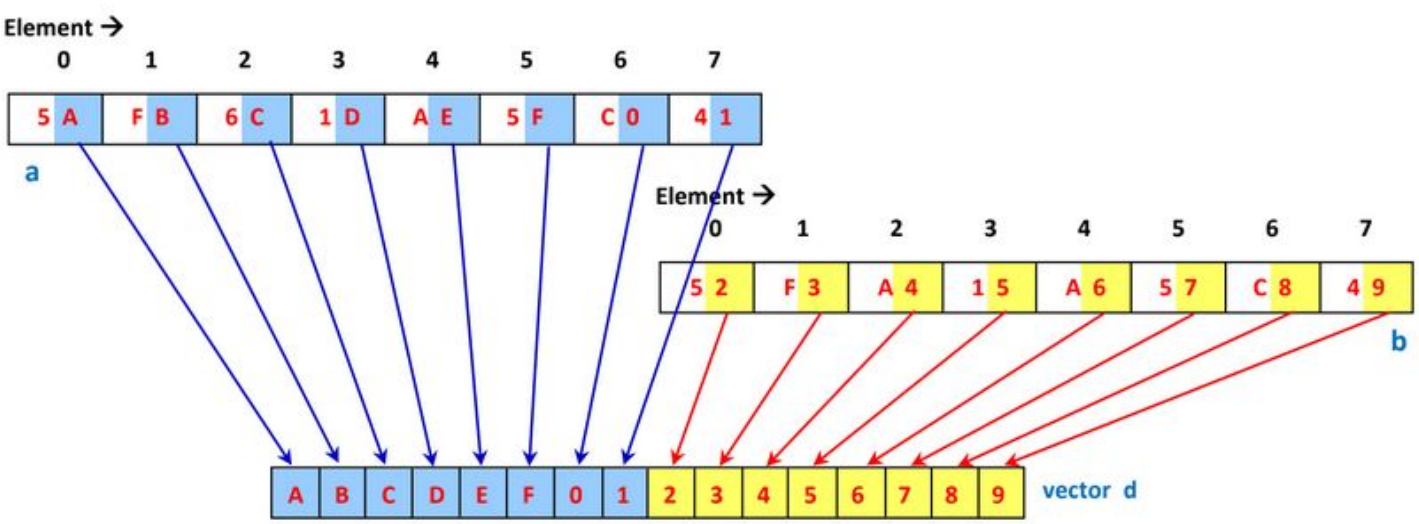
31 – 28	27 – 20	19 – 15	14 – 10	9 – 5	4 – 0
0111	00001000	rs	rt	00000	rd
Constant	Opcode	rs	rt	ru	rd

Syntax: vec_pack d, a, b

Vectors Used: a, b, d

Description: The wider elements in vector a are truncated together to form the high corresponding elements in vector d; while the wider elements in vector b are truncated together to form the corresponding elements in vector d.

Representation:



7.10 Vector Permute

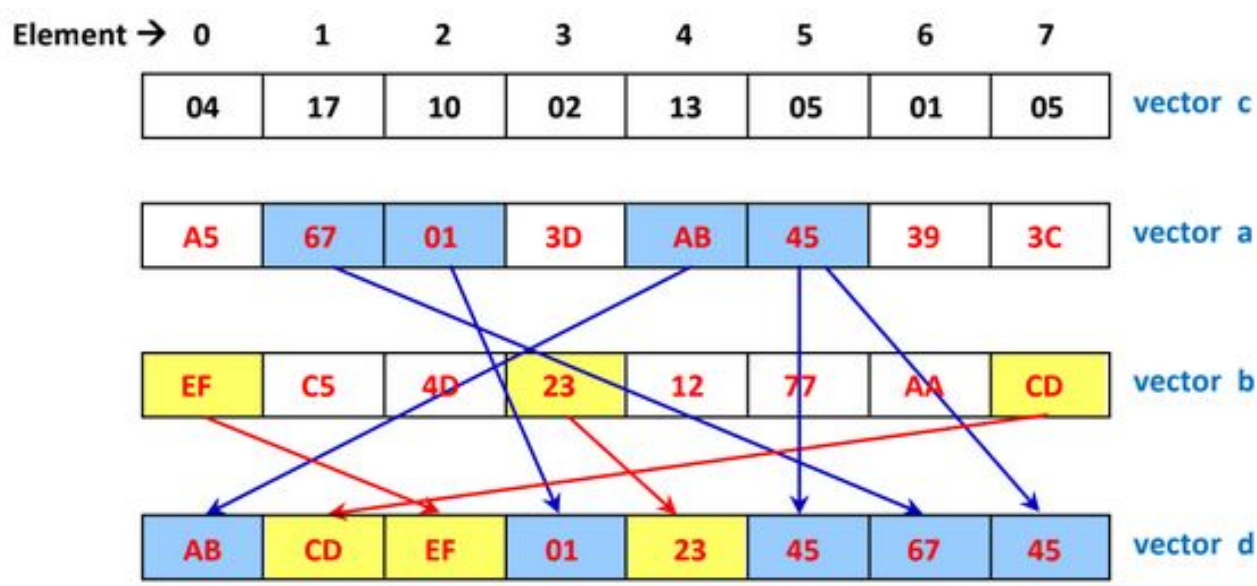
31 – 28	27 – 20	19 – 15	14 – 10	9 – 5	4 – 0
0111	00001001	rs	rt	ru	rd
Constant	Opcode	rs	rt	ru	rd

Syntax: vec_perm d, a, b,c

Vectors Used: a, b, c, d

Description: Each element in vector c specifies which elements to be stored the corresponding element in vector d. The most-significant half of each element in vector c specifies which vector to choose from (0 = a or 1 = b). The least significant half of each element in vector c specifies the element of the vector to be selected

Representation:



7.11 Vector Compare Equal-To

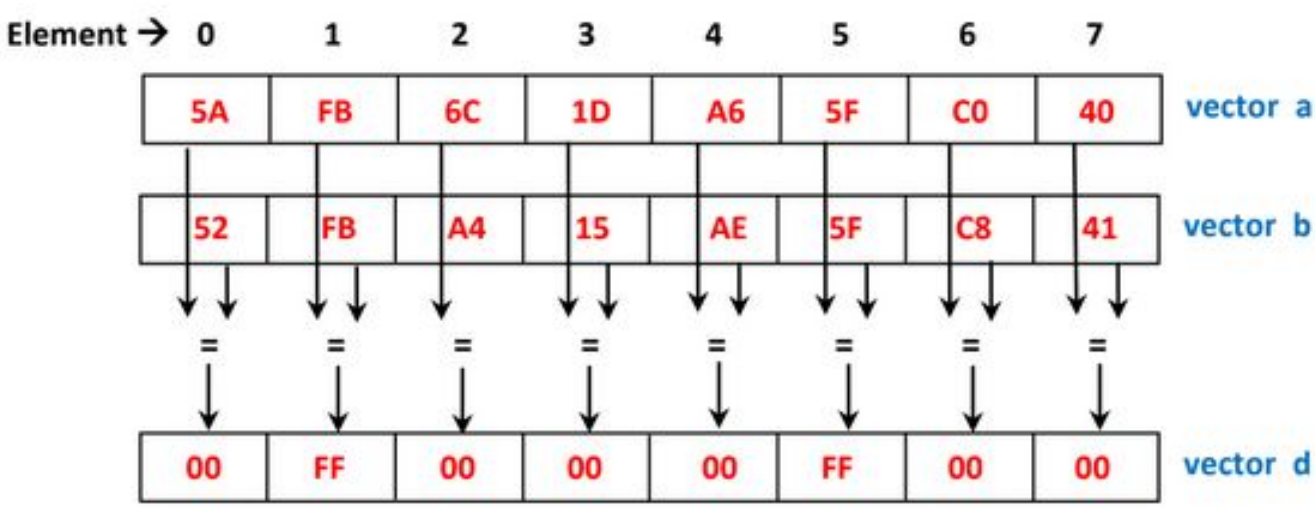
31 – 28	27 – 20	19 – 15	14 – 10	9 – 5	4 – 0
0111	00001010	rs	rt	00000	rd
Constant	Opcode	rs	rt	ru	rd

Syntax: vec_cmpeq d, a, b

Vectors Used: a, b, d

Description: Compare each element in vector a and the corresponding element in vector b. If the element in vector a equals to the corresponding element in vector b, thebits in the corresponding element in vector d will be set to all 1. Otherwise, the bits in the corresponding element in vector d will be set to all 0

Representation:



7.12 Vector Compare Less-Than (unsigned)

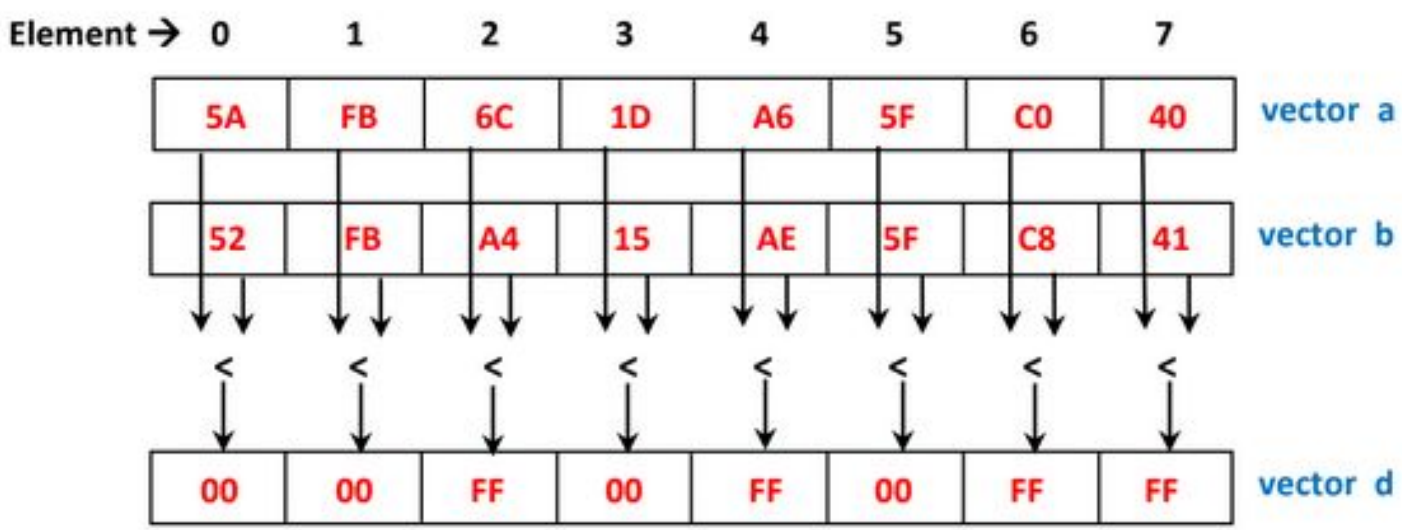
31 – 28	27 – 20	19 – 15	14 – 10	9 – 5	4 – 0
0111	00001011	rs	rt	00000	rd
Constant	Opcode	rs	rt	ru	rd

Syntax: vec_cmpltu d, a, b

Vectors Used: a, b, d

Description: Compare each element in vector a and the corresponding element in vector b. If the element in vector a is less than the corresponding element in vector b, the bits in the corresponding element in vector d will be set to all 1. Otherwise, the bits in the corresponding element in vector d will be set to all 0

Representation:



7.13 Vector And

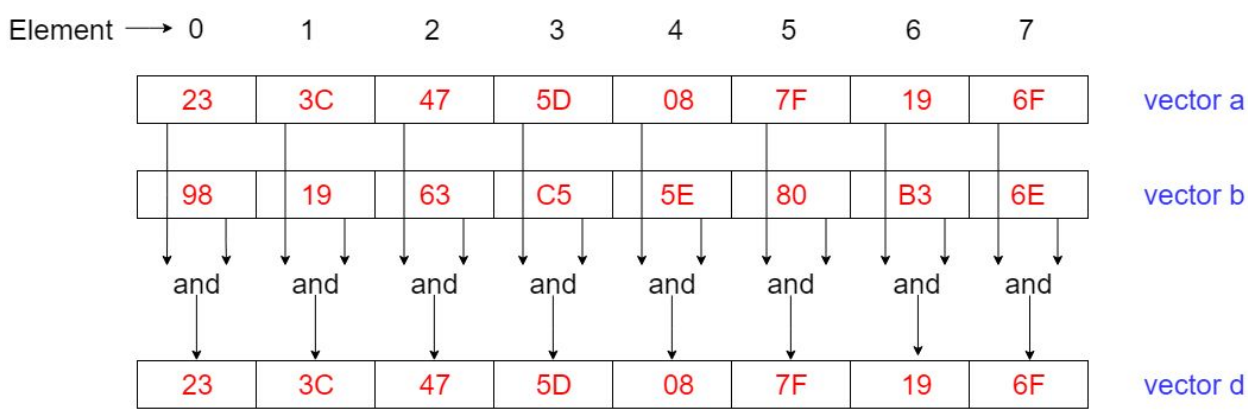
31 – 28	27 – 20	19 – 15	14 – 10	9 – 5	4 – 0
0111	00001100	rs	rt	00000	rd
Constant	Opcode	rs	rt	ru	rd

Syntax: vec_and d, a, b

Vectors Used: a, b, d

Description: Perform a bitwise-and operation on each element from vector a and vector b, one at a time. The result is stored into the corresponding element of vector d.

Representation:



7.14 Vector Branch if Equal

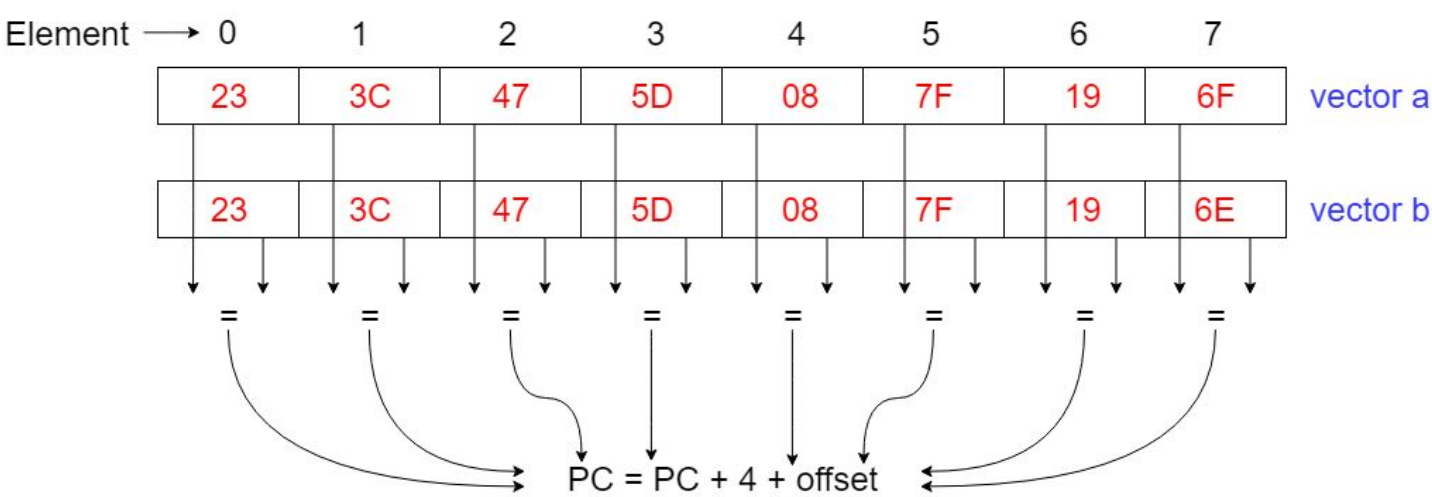
31 – 28	27 – 20	19 – 15	14 – 10	9 – 0
0111	0x00001101	rs	rt	offset
Constant	Opcode	rs	rt	imm

Syntax: vec_beq a, b, imm

Vectors Used: a, b

Description: Compare each element in vector a and the corresponding element in vector b. If each element of a is equal to the corresponding element of b, branch to the address specified by the immediate value. It uses PC offset format to determine the address.

Representation:



7.15 Vector Branch if not Equal

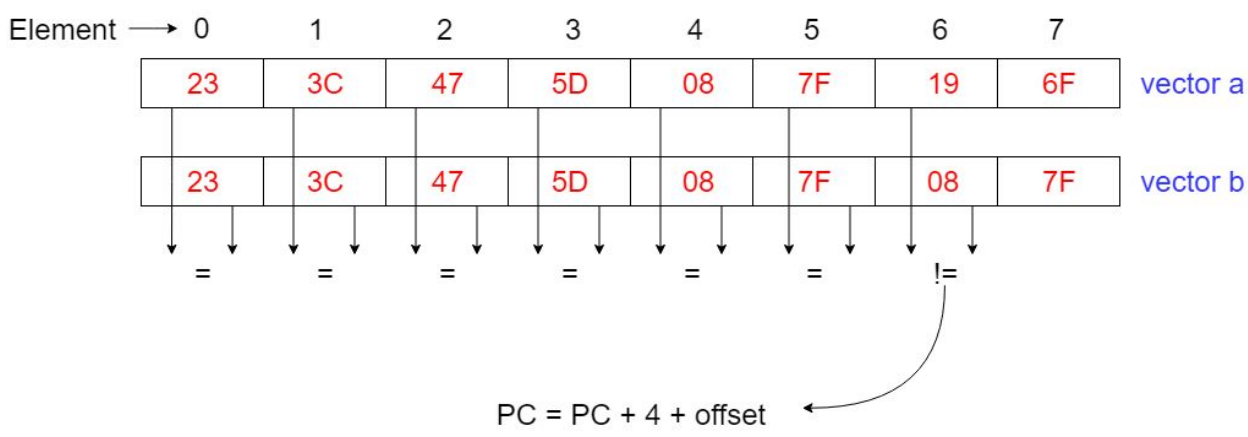
31 – 28	27 – 20	19 – 15	14 – 10	9 – 0
0111	0x00001110	rs	rt	offset
Constant	Opcode	rs	rt	im

Syntax: vec_bne a, b, imm

Vectors Used: a, b

Description: Compare each element in vector a and the corresponding element in vector b. If any element of a isn't equal to the corresponding element of b, branch to the address specified by the immediate value. It uses PC offset format to determine the address.

Representation:



7.16 Vector Decrypt

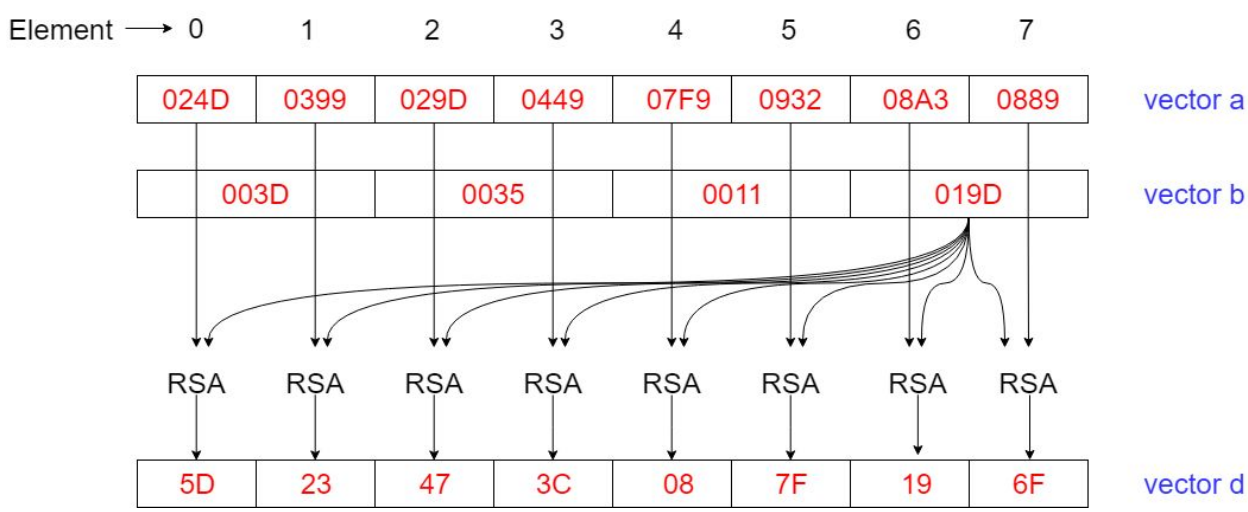
31 – 28	27 – 20	19 – 15	14 – 10	9 – 5	4 – 0
0111	00001111	rs	rt	00000	rd
Constant	Opcode	rs	rt	ru	rd

Syntax: vec_decry d, a, b

Vectors Used: a, b, d

Description: Vectors a and b are unique. a is an eight-element vector of sixteen bytes. b is a four-element vector of eight bytes. Each element of a is decrypted using RSA and stored into the elements of d. p, q, and d are provided by the elements of d.

Representation:



7.17 Vector Encrypt

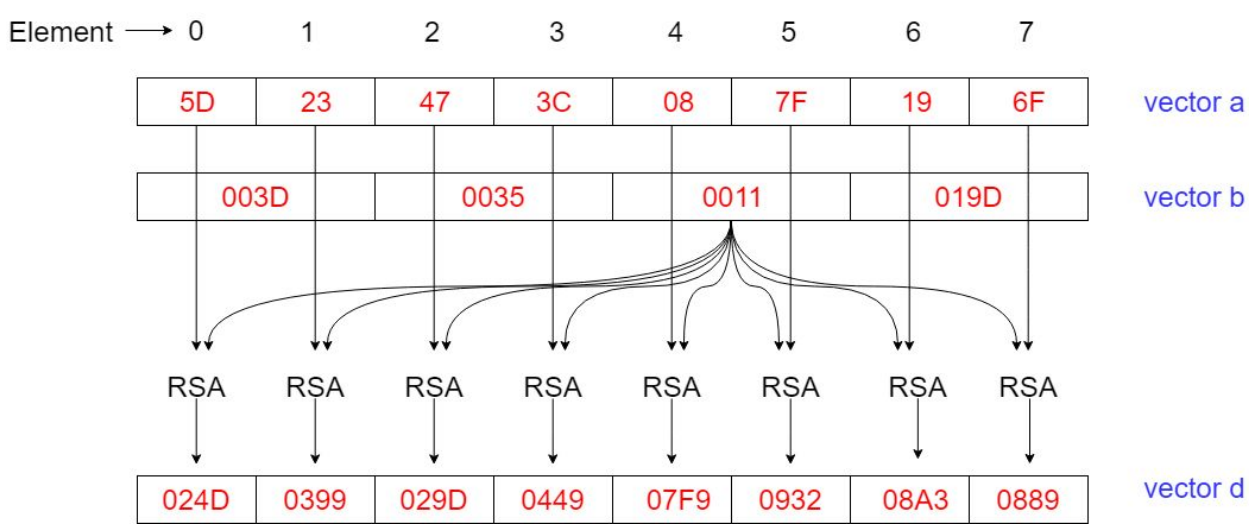
31 – 28	27 – 20	19 – 15	14 – 10	9 – 5	4 – 0
0111	00010000	rs	rt	00000	rd
Constant	Opcode	rs	rt	ru	rd

Syntax: vec_encry d, a, b

Vectors Used: a, b, d

Description: Vectors d and b are unique. d is an eight-element vector of sixteen bytes. b is a four-element vector of eight bytes. Each element of a is encrypted using RSA and stored into the elements of d. p, q, and e are provided by the elements of d.

Representation:



7.18 Vector Or

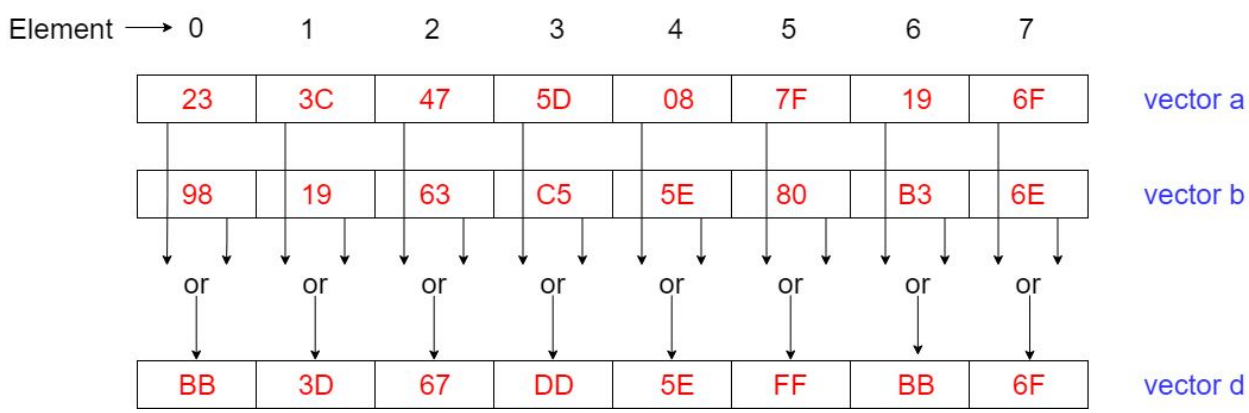
31 – 28	27 – 20	19 – 15	14 – 10	9 – 5	4 – 0
0111	00010001	rs	rt	00000	rd
Constant	Opcode	rs	rt	ru	rd

Syntax: vec_or d, a, b

Vectors Used: a, b, d

Description: Perform a bitwise-or operation on each element from vector a and vector b, one at a time. The result is stored into the corresponding element of vector d.

Representation:



7.19 Vector Sort Descending

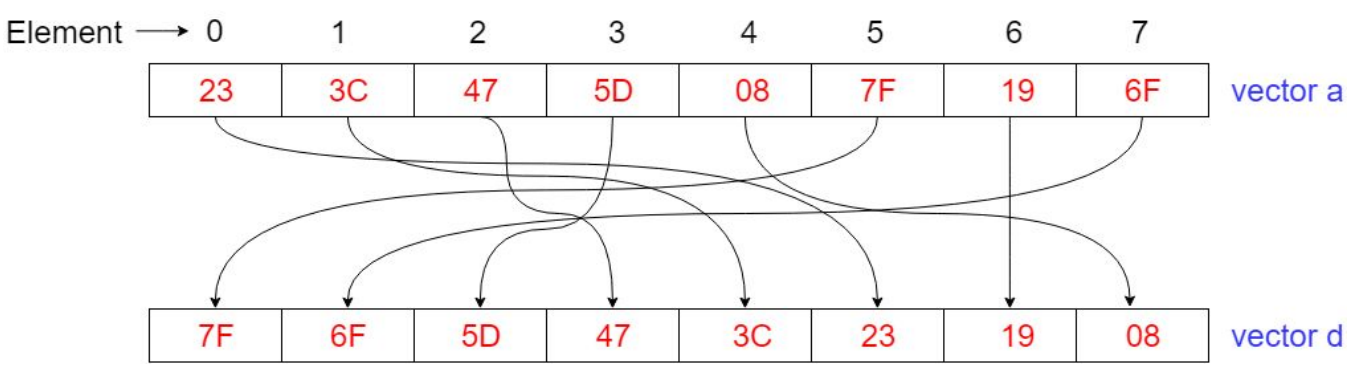
31 – 28	27 – 20	19 – 15	14 – 10	9 – 5	4 – 0
0111	00010010	rs	00000	00000	rd
Constant	Opcode	rs	rt	ru	rd

Syntax: vec_sortlow d, a

Vectors Used: a, d

Description: Sorts the elements in vector a from greatest value to smallest value. The new sorted vector is stored into vector d.

Representation:



7.20 Vector Sort Ascending

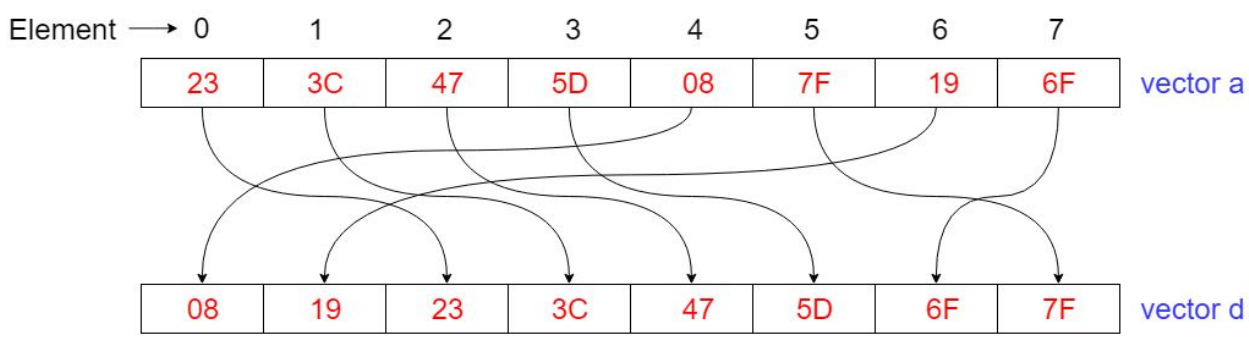
31 – 28	27 – 20	19 – 15	14 – 10	9 – 5	4 – 0
0111	00010011	rs	00000	00000	rd
Constant	Opcode	rs	rt	ru	rd

Syntax: vec_sorthi d, a

Vectors Used: a, d

Description: Sorts the elements in vector a from least value to greatest value. The new sorted vector is stored into vector d.

Representation:



7.21 Vector Subtract Saturated

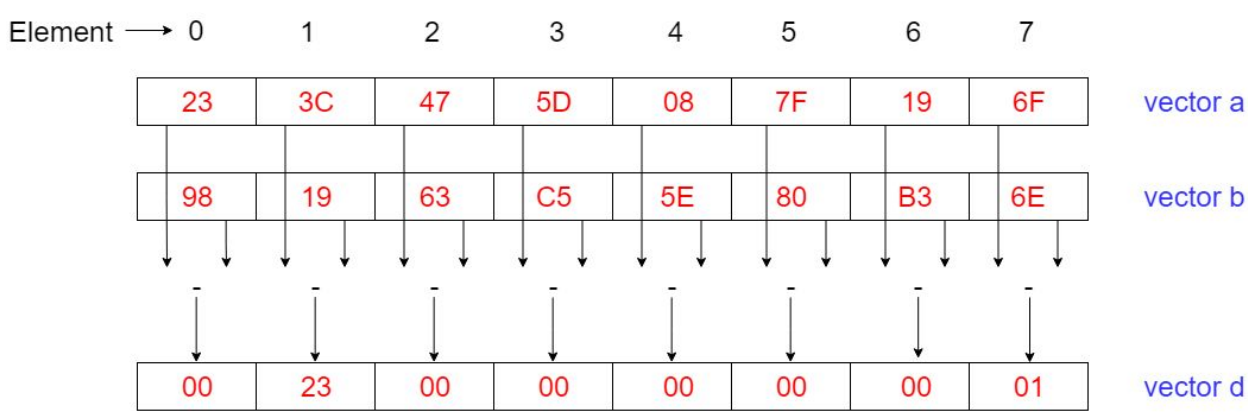
31 – 28	27 – 20	19 – 15	14 – 10	9 – 5	4 – 0
0111	00001100	rs	rt	00000	rd
Constant	Opcode	rs	rt	ru	rd

Syntax: vec_subsu d, a, b

Vectors Used: a, b, c

Description: The element in vector a is subtracted from the element in vector b. The result is stored into the corresponding element of vector d. This is repeated for all eight vectors. If an operation would result in a negative number, zero is stored into d instead.

Representation:



E. Summary

A total of 21 SIMD instructions are implemented in this manual. 12 Baseline SIMD instructions and 9 Application Specific instructions.

Baseline SIMD	**Application Specific**
vec_addsu	vec_and
vec_madd	vec_beq
vec_mule	vec_bne
vec_mulo	vec_decry
vec_msums	vec_encry
vec_splat	vec_or
vec_mergel	vec_sortlow
vec_mergeh	vec_sortup
vec_pack	vec_subsu
vec_perm	
vec_cmpeq	
vec_cmpltu	

Figure: Enhancement List

1. Baseline SIMD Enhancements (Instruction Format)

1.1 Vec_addsu

31 28	27 20	19 15	14 10	9 5	4 0
0111	00000000	rs	rt	00000	rd
Constant	Opcode	rs	rt	ru	rd

1.2 Vec_madd

31 28	27 20	19 15	14 10	9 5	4 0
0111	00000001	rs	rt	ru	rd
Constant	Opcode	rs	rt	ru	rd

1.3 Vec_mule

31 28	27 20	19 15	14 10	9 5	4 0
0111	00000010	rs	rt	00000	rd
Constant	Opcode	rs	rt	ru	rd

1.4 Vec_mulo

31 28	27 20	19 15	14 10	9 5	4 0
0111	00000011	rs	rt	00000	rd
Constant	Opcode	rs	rt	ru	rd

1.5 Vec_msums

31 28	27 20	19 15	14 10	9 5	4 0
0111	00000100	rs	rt	ru	rd
Constant	Opcode	rs	rt	ru	rd

1.6 Vec_splat

31 28	27 20	19 15	14 10	9 5	4 0
0111	00000101	rs	rt	00000	rd
Constant	Opcode	rs	rt	ru	rd

1.7 Vec_mergel

31 28	27 20	19 15	14 10	9 5	4 0
0111	00000110	rs	rt	00000	rd
Constant	Opcode	rs	rt	ru	rd

1.8 Vec_mergeh

31 28	27 20	19 15	14 10	9 5	4 0
0111	00000111	rs	rt	00000	rd
Constant	Opcode	rs	rt	ru	rd

1.9 Vec_pack

31 28	27 20	19 15	14 10	9 5	4 0
0111	00001000	rs	rt	00000	rd
Constant	Opcode	rs	rt	ru	rd

1.10 Vec_perm

31 28	27 20	19 15	14 10	9 5	4 0
0111	00001001	rs	rt	ru	rd
Constant	Opcode	rs	rt	ru	rd

1.11 Vec_cmpeq

31 28	27 20	19 15	14 10	9 5	4 0
0111	00001010	rs	rt	00000	rd
Constant	Opcode	rs	rt	ru	rd

1.12 Vec_cmpltu

31 28	27 20	19 15	14 10	9 5	4 0
0111	00001011	rs	rt	00000	rd
Constant	Opcode	rs	rt	ru	rd

2. Application Specific Enhancements (Instruction Format)

2.1 Vec_and

31 – 28	27 – 20	19 – 15	14 – 10	9 – 5	4 – 0
0111	00001100	rs	rt	00000	rd
Constant	Opcode	rs	rt	ru	rd

2.2 Vec_beq

31 – 28	27 – 20	19 – 15	14 – 10	9 – 0
0111	0x00001101	rs	rt	offset
Constant	Opcode	rs	rt	imm

2.3 Vec_bne

31 – 28	27 – 20	19 – 15	14 – 10	9 – 0
0111	0x00001110	rs	rt	offset
Constant	Opcode	rs	rt	im

2.4 Vec_decry

31 28	27 20	19 15	14 10	9 5	4 0
0111	00001111	rs	rt	00000	rd
Constant	Opcode	rs	rt	ru	rd

2.5 Vec_encry

31 28	27 20	19 15	14 10	9 5	4 0
0111	00010000	rs	rt	00000	rd
Constant	Opcode	rs	rt	ru	rd

2.6 Vec_or

31 28	27 20	19 15	14 10	9 5	4 0
0111	00010001	rs	rt	00000	rd
Constant	Opcode	rs	rt	ru	rd

2.7 Vec_sortlow

31 28	27 20	19 15	14 10	9 5	4 0
0111	00010010	rs	00000	00000	rd
Constant	Opcode	rs	rt	ru	rd

2.8 Vec_sortup

31 28	27 20	19 15	14 10	9 5	4 0
0111	00010011	rs	00000	00000	rd
Constant	Opcode	rs	rt	ru	rd

2.9 Vec_subsu

31 28	27 20	19 15	14 10	9 5	4 0
0111	00001100	rs	rt	00000	rd
Constant	Opcode	rs	rt	ru	rd

III. MIPS Implementation / Verification with Annotation

A. Source code

Listed below are the source codes of all the MIPS coded modules for all components from which the processor is constructed.

Baseline SIMD	Application Specific
vec_addsu	vec_and
vec_madd	vec_beq
vec_mule	vec_bne
vec_mulo	vec_decry
vec_msums	vec_encry
vec_splat	vec_or
vec_mergel	vec_sortlow
vec_mergeh	vec_sortup
vec_pack	vec_subsu
vec_perm	
vec_cmpeq	
vec_cmpltu	

Figure: Enhancement List

1. Baseline SIMD Enhancements

1.1 Vec_addsu

```
#*******************************************************************************
# File name:          vec_addsu.asm
# Version:            1.0
# Date:               November 12, 2018
# Programmer:         Ellen Burger
#
# Description:        Code implementing the instruction 'vec_addsu d, a, b' into MIPS
#                     architecture. D,A, and B are 8-byte vectors with 2-byte wide
#                      elements.The upper and lower halves of each vector are split between
#                      two registers. The instruction adds each element from A and B and
#                      stores it into the corresponding element in vector D.
#
# Register usage:     $t0 - contains the upper half of vector A
#                     $t1 - contains the lower half of vector A
#                     $t2 - contains the upper half of vector B
#                     $t3 - contains the lower half of vector B
#                     $t4 - contains vector A's two bytes that are being added, then receives
#                    the sum to be stored into vector D. Set to 0xff if the sum causes
#                    overflow
#                     $t5 - contains vector B's two bytes that are being added
#                     $s0 - contains the upper half of vector D
#                     $s1 - contains the lower half of vector D
#                     $a0 - set to 0 if there is a sum overflow
#                     $ra - holds the return address
#*******************************************************************************

# Main code segment
.text
.globl  main

main:  li            $t0, 0x233C475D       # load the lower half of vector A into t0
       li            $t1, 0x087F196F       # load the upper half of vector A into t1
       li            $t2, 0x981963C5       # load the lower half of vector B into t2
       li            $t3, 0x5E80B36E       # load the upper half of vector B into t3
```

```
#       Upper half of the vectors
        srl             $t4, $t0, 24          # shift t4 and t5 right to gain their first
                                              #elements
        srl             $t5, $t2, 24
        add             $t4, $t4, $t5         # add elements and store the result in t4
        jal             Check                 # jump to subroutine to check for overflow
        add             $s0, $t4, $zero       # store sum into s0, the destination vector D
        sll             $s0, $s0, 8           # shift the bytes twice to make room for the next
                                              # sum
        sll             $t4, $t0, 8           # shift t4 and t5 left then right to gain their
                                              # 2nd elements
        sll             $t5, $t2, 8
        srl             $t4, $t4, 24
        srl             $t5, $t5, 24
        add             $t4, $t4, $t5         # add elements and store the result in t4
        jal             Check                 # jump to subroutine to check for overflow
        add             $s0, $s0, $t4         # store sum into s0, the destination vector D
        sll             $s0, $s0, 8           # shift the bytes twice to make room for the next
                                              # sum

        sll             $t4, $t0, 16          # shift t4 and t5 left then right to gain their
                                              # 3rd elements
        sll             $t5, $t2, 16
        srl             $t4, $t4, 24
        srl             $t5, $t5, 24
        add             $t4, $t4, $t5         # add elements and store the result in t4
        jal             Check                 # jump to subroutine to check for overflow
        add             $s0, $s0, $t4         # store sum into s0, the destination vector D
        sll             $s0, $s0, 8           # shift the bytes twice to make room for the next
                                              # sum

        sll             $t4, $t0, 24          # shift t4 and t5 left then right to gain their
                                              # 4th elements
        sll             $t5, $t2, 24
        srl             $t4, $t4, 24
        srl             $t5, $t5, 24
        add             $t4, $t4, $t5         # add elements and store the result in t4
        jal             Check                 # jump to subroutine to check for overflow
        add             $s0, $s0, $t4         # store sum into s0, the destination vector D

#       Lower half of the vectors
        srl             $t4, $t1, 24          # shift t4 and t5 right to gain their fifth
                                              # elements
        srl             $t5, $t3, 24
        add             $t4, $t4, $t5         # add elements and store the result in t4
        jal             Check                 # jump to subroutine to check for overflow
        add             $s1, $t4, $zero       # store sum into s1, the destination vector D
        sll             $s1, $s1, 8           # shift the bytes twice to make room for the next
```

```
                                        # sum

        sll         $t4, $t1, 8         # shift t4 and t5 left then right to gain their
                                        # 6th elements
        sll         $t5, $t3, 8
        srl         $t4, $t4, 24
        srl         $t5, $t5, 24
        add         $t4, $t4, $t5       # add elements and store the result in t4
        jal         Check               # jump to subroutine to check for overflow
        add         $s1, $s1, $t4       # store sum into s1, the destination vector D
        sll         $s1, $s1, 8         # shift the bytes twice to make room for the next
                                        # sum

        sll         $t4, $t1, 16        # shift t4 and t5 left then right to gain their
                                        # 7th elements
        sll         $t5, $t3, 16
        srl         $t4, $t4, 24
        srl         $t5, $t5, 24
        add         $t4, $t4, $t5       # add elements and store the result in t4
        jal         Check               # jump to subroutine to check for overflow
        add         $s1, $s1, $t4       # store sum into s1, the destination vector D
        sll         $s1, $s1, 8         # shift the bytes twice to make room for the next
                                        # sum

        sll         $t4, $t1, 24        # shift t4 and t5 left then right to gain their
                                        # 8th elements
        sll         $t5, $t3, 24
        srl         $t4, $t4, 24
        srl         $t5, $t5, 24
        add         $t4, $t4, $t5       # add elements and store the result in t4
        jal         Check               # jump to subroutine to check for overflow
        add         $s1, $s1, $t4       # store sum into s1, the destination vector D

#       Function code for exit
exit:   ori         $v0, $zero, 10
        syscall

#       Subroutine to check for overflow
Check:  slti        $a0, $t4, 0xFF      # check if t4 is greater than 0xFF causing
                                        # overflow
        bne         $a0, $zero, End     # exits subroutine if not
        addi        $t4, $zero, 0xFF    # sets $t4 to 0xFF is there's overflow
End:    jr          $ra                 # return to main routine
```

1.2 Vec_madd

```
#*******************************************************************************
# File name:          vec_madd.asm
# Version:            1.0
# Date:               November 12, 2018
# Programmer:         Ellen Burger
#
# Description:        Code implementing the instruction 'vec_madd d, a, b, c' into
#                      MIPS architecture.
#                      D, A, B, and C are 8-byte vectors with 2-byte wide elements. The upper
#                      and lower halves of each vector are split between two registers. The
#                      instruction multiplies the first elements from A and B then adds the
#                      product with the element in C, then stores it into the corresponding
#                      element in vector D. This is repeated for all eight elements.
#
# Register usage:     $t0 - contains the upper half of vector A
#                     $t1 - contains the lower half of vector A
#                     $t2 - contains the upper half of vector B
#                     $t3 - contains the lower half of vector B
#                     $t4 - contains the lower half of vector C
#                     $t5 - contains the upper half of vector C
#                     $t6 - contains vector A's two bytes that are being multiplied. It holds
#                      the final sum of the operations and stores into vector D
#                     $t7 - contains vector B's two bytes that are being multiplied. It then
#                      holds the element in vector C in the latter half of each operation
#                     $s0 - contains the upper half of vector D
#                     $s1 - contains the lower half of vector D
#                     $a0 - set to 0 if there is a sum overflow
#                     $ra - holds the return address
#*******************************************************************************

# Main code segment
.text
.globl  main

main:   li          $t0, 0x120C1A0D             # load the lower half of vector A into t0
        li          $t1, 0x23051912             # load the upper half of vector A into t1
        li          $t2, 0x3D0C104D             # load the lower half of vector B into t2
        li          $t3, 0x057F192B             # load the upper half of vector B into t3
```

```
        li          $t4, 0x60091B05         # load the lower half of vector C into t8
        li          $t5, 0x501E0660         # load the upper half of vector C into t9

#       Upper half of the vectors
        srl         $t6, $t0, 24        # shift t6 and t7 right to gain their first
                                        # elements
        srl         $t7, $t2, 24
        mult        $t6, $t7            # multiply elements
        mflo        $t6                 # store the result into t6
        srl         $t7, $t4, 24        # shift t7 to gain C's first element
        add         $t6, $t6, $t7       # add the product to t7
        jal         Check               # jump to subroutine to check for overflow
        add         $s0, $s0, $t6       # store sum into s0, the destination vector D
        sll         $s0, $s0, 8         # shift the bytes two times to make room for the
                                        # next sum
        sll         $t6, $t0, 8         # shift t6 and t7 left then right to gain their
                                        # second elements
        sll         $t7, $t2, 8
        srl         $t6, $t6, 24
        srl         $t7, $t7, 24
        mult        $t6, $t7            # multiply elements
        mflo        $t6                 # store the result into t6
        sll         $t7, $t4, 8
        srl         $t7, $t7, 24        # shift t7 to gain C's second element
        add         $t6, $t6, $t7       # add the product to t7
        jal         Check               # jump to subroutine to check for overflow
        add         $s0, $s0, $t6       # store sum into s0, the destination vector D
        sll         $s0, $s0, 8         # shift the bytes two times to make room for the
                                        # next sum
        sll         $t6, $t0, 16        # shift t6 and t7 left then right to gain their
                                        # third elements
        sll         $t7, $t2, 16
        srl         $t6, $t6, 24
        srl         $t7, $t7, 24
        mult        $t6, $t7            # multiply elements
        mflo        $t6                 # store the result into t6
        sll         $t7, $t4, 16
        srl         $t7, $t7, 24        # shift t7 to gain C's third element
        add         $t6, $t6, $t7       # add the product to t7
        jal         Check               # jump to subroutine to check for overflow
        add         $s0, $s0, $t6       # store sum into s0, the destination vector D
        sll         $s0, $s0, 8         # shift the bytes two times to make room for the
                                        # next sum
        sll         $t6, $t0, 24        # shift t6 and t7 left then right to gain their
                                        # fourth elements
        sll         $t7, $t2, 24
        srl         $t6, $t6, 24
        srl         $t7, $t7, 24
```

```
        mult        $t6, $t7            # multiply elements
        mflo        $t6                 # store the result into t6
        sll         $t7, $t4, 24
        srl         $t7, $t7, 24        # shift t7 to gain C's fourth element
        add         $t6, $t6, $t7       # add the product to t7
        jal         Check               # jump to subroutine to check for overflow
        add         $s0, $s0, $t6       # store sum into s0, the destination vector D

#       Lower half of the vectors
        srl         $t6, $t1, 24        # shift t6 and t7 right to gain their fifth
                                        # elements
        srl         $t7, $t3, 24
        mult        $t6, $t7            # multiply elements
        mflo        $t6                 # store the result into t6
        srl         $t7, $t5, 24        # shift t7 to gain C's fifth element
        add         $t6, $t6, $t7       # add the product to t7
        jal         Check               # jump to subroutine to check for overflow
        add         $s1, $s1, $t6       # store sum into s1, the destination vector D
        sll         $s1, $s1, 8         # shift the bytes two times to make room for the
                                        # next sum
        sll         $t6, $t1, 8         # shift t6 and t7 left then right to gain their
                                        # sixth elements
        sll         $t7, $t3, 8
        srl         $t6, $t6, 24
        srl         $t7, $t7, 24
        mult        $t6, $t7            # multiply elements
        mflo        $t6                 # store the result into t6
        sll         $t7, $t5, 8
        srl         $t7, $t7, 24        # shift t7 to gain C's sixth element
        add         $t6, $t6, $t7       # add the product to t7
        jal         Check               # jump to subroutine to check for overflow
        add         $s1, $s1, $t6       # store sum into s1, the destination vector D
        sll         $s1, $s1, 8         # shift the bytes two times to make room for the
                                        # next sum
        sll         $t6, $t1, 16        # shift t6 and t7 left then right to gain their
                                        # seventh elements
        sll         $t7, $t3, 16
        srl         $t6, $t6, 24
        srl         $t7, $t7, 24
        mult        $t6, $t7            # multiply elements
        mflo        $t6                 # store the result into t6
        sll         $t7, $t5, 16
        srl         $t7, $t7, 24        # shift t7 to gain C's seventh element
        add         $t6, $t6, $t7       # add the product to t7
        jal         Check               # jump to subroutine to check for overflow
        add         $s1, $s1, $t6       # store sum into s1, the destination vector D
        sll         $s1, $s1, 8         # shift the bytes two times to make room for the
```

```
                                        # next sum
        sll         $t6, $t1, 24        # shift t6 and t7 left then right to gain their
                                        # eighth elements
        sll         $t7, $t3, 24
        srl         $t6, $t6, 24
        srl         $t7, $t7, 24
        mult        $t6, $t7            # multiply elements
        mflo        $t6                 # store the result into t6
        sll         $t7, $t5, 24
        srl         $t7, $t7, 24        # shift t7 to gain C's eighth element
        add         $t6, $t6, $t7       # add the product to t7
        jal         Check               # jump to subroutine to check for overflow
        add         $s1, $s1, $t6       # store sum into s1, the destination vector D

#       Function code for exit
exit:   ori         $v0, $zero, 10
        syscall

#       Subroutine to check for overflow
Check:  slti        $a0, $t6, 0xFF      # check if t6 is greater than 0xFF causing
                                        # overflow
        bne         $a0, $zero, End     # exits subroutine if not
        sll         $t6, $t6, 24        # shifts t6 left then right so only the two least
                                        # significant bits are present if so
        srl         $t6, $t6, 24
End:    jr          $ra                 # return to main routine
```

1.3 Vec_mule

```
#*******************************************************************************
# File name:          vec_mule.asm
# Version:            1.0
# Date:               November 12, 2018
# Programmer:         Ellen Burger
#
# Description:        Code implementing the instruction 'vec_mule d, a, b' into MIPS
#                      architecture. A, B, and C are 8-byte vectors with 2-byte wide elements,
#                      and D has 4-byte wide elements. The upper and lower halves of each
#                      vector are split between two registers. The instruction multiplies the
#                      even-numbered element positions of A and B and stores the 16-bit
#                      product into vector D.
#
# Register usage:     $t0 - contains the upper half of vector A
#                     $t1 - contains the lower half of vector A
#                     $t2 - contains the upper half of vector B
#                     $t3 - contains the lower half of vector B
#                     $t4 - contains vector A's two bytes that are being multiplied, handling
#                      the even-numbered element positions. It holds the product and stores it
#                      into vector D.
#                     $t5 - contains vector B's two bytes that are being multiplied, handling
#                      the even-numbered element positions.
#                     $s0 - contains the upper half of vector D
#                     $s1 - contains the lower half of vector D
#                     $a0 - set to 0 if there is a product overflow
#                     $a1 - # set to 0xFFFF to check for overflows
#                     $ra - holds the return address
#*******************************************************************************

# Main code segment
.text
.globl  main

main:   li          $t0, 0xAEE95AE0       # load the lower half of vector A into t0
        li          $t1, 0xF080CC66       # load the upper half of vector A into t1
        li          $t2, 0x33146170       # load the lower half of vector B into t2
        li          $t3, 0x609888AB       # load the upper half of vector B into t3
        li          $a1, 0xFFFF           # set a1 to 0xFFFF

#       Upper half of the vectors
```

```
        srl         $t4, $t0, 24        # shift t4 and t5 right to gain their first
elements
        srl         $t5, $t2, 24
        mult        $t4, $t5            # multiply elements
        mflo        $t4                 # store the result into t4
        jal         Check               # jump to subroutine to check for overflow
        add         $s0, $s0, $t4       # store product into s0, the destination vector D
        sll         $s0, $s0, 16        # shift the bytes four times to make room for the
                                        # next sum

        sll         $t4, $t0, 16        # shift t4 and t5 left then right to gain their
                                        # third elements
        sll         $t5, $t2, 16
        srl         $t4, $t4, 24
        srl         $t5, $t5, 24
        mult        $t4, $t5            # multiply elements
        mflo        $t4                 # store the result into t4
        jal         Check               # jump to subroutine to check for overflow
        add         $s0, $s0, $t4       # store product into s0, the destination vector D

#       Lower half of the vectors
        srl         $t4, $t1, 24        # shift t4 and t5 right to gain their fifth
                                        # elements
        srl         $t5, $t3, 24
        mult        $t4, $t5            # multiply elements
        mflo        $t4                 # store the result into t4
        jal         Check               # jump to subroutine to check for overflow
        add         $s1, $s1, $t4       # store product into s1, the destination vector D
        sll         $s1, $s1, 16        # shift the bytes four times to make room for the
                                        # next sum

        sll         $t4, $t1, 16        # shift t4 and t5 left then right to gain their
                                        # seventh elements
        sll         $t5, $t3, 16
        srl         $t4, $t4, 24
        srl         $t5, $t5, 24
        mult        $t4, $t5            # multiply elements
        mflo        $t4                 # store the result into t4
        jal         Check               # jump to subroutine to check for overflow
        add         $s1, $s1, $t4       # store product into s1, the destination vector D

#       Function code for exit
exit:   ori         $v0, $zero, 10
        syscall

#       Subroutine to check for overflow
Check:  slt         $a0, $t4, $a1       # check if t4 is greater than 0xFFFF causing
```

```
                                        # overflow
        bne         $a0, $zero, End     # exits subroutine if not
        add         $t4, $zero, $a1     # sets $t4 to 0xFFFF is there's overflow
End:    jr          $ra                 # return to main routine
```

1.4 Vec_mulo

```
#*******************************************************************************
# File name:          vec_mulo.asm
# Version:            1.0
# Date:               November 12, 2018
# Programmer:         Ellen Burger
#
# Description:        Code implementing the instruction 'vec_mulo d, a, b' into MIPS
#                      architecture. A, B, and C are 8-byte vectors with 2-byte wide elements,
#                      and D has 4-byte wide elements. The upper and lower halves of each
#                      vector are split between two registers. The instruction multiplies the
#                      odd-numbered element positions of A and B and stores the 16-bit product
#                      into vector D.
#
# Register usage:     $t0 - contains the upper half of vector A
#                     $t1 - contains the lower half of vector A
#                     $t2 - contains the upper half of vector B
#                     $t3 - contains the lower half of vector B
#                     $t4 - contains vector A's two bytes that are being multiplied, handling
#                      the odd-numbered element positions. It holds the product and stores it
#                      into vector D.
#                     $t5 - contains vector B's two bytes that are being multiplied, handling
#                      the odd-numbered element positions.
#                     $s0 - contains the upper half of vector D
#                     $s1 - contains the lower half of vector D
#                     $a0 - set to 0 if there is a product overflow
#                     $a1 - # set to 0xFFFF to check for overflows
#                     $ra - holds the return address
#*******************************************************************************

# Main code segment
.text
.globl  main

main:  li            $t0, 0xAEE95AE0       # load the lower half of vector A into t0
       li            $t1, 0xF080CC66       # load the upper half of vector A into t1
       li            $t2, 0x33146170       # load the lower half of vector B into t2
       li            $t3, 0x609888AB       # load the upper half of vector B into t3
       li            $a1, 0xFFFF           # set a1 to 0xFFFF
```

```
#       Upper half of the vectors
        sll             $t4, $t0, 8          # shift t4 and t5 left then right to gain their
                                             # second elements
        sll             $t5, $t2, 8
        srl             $t4, $t4, 24
        srl             $t5, $t5, 24
        mult            $t4, $t5             # multiply elements
        mflo            $t4                  # store the result into t4
        jal             Check                # jump to subroutine to check for overflow
        add             $s0, $s0, $t4        # store product into s0, the destination vector D
        sll             $s0, $s0, 16         # shift the bytes four times to make room for the
                                             # next sum

        sll             $t4, $t0, 24         # shift t4 and t5 left then right to gain their
                                             # fourth elements
        sll             $t5, $t2, 24
        srl             $t4, $t4, 24
        srl             $t5, $t5, 24
        mult            $t4, $t5             # multiply elements
        mflo            $t4                  # store the result into t4
        jal             Check                # jump to subroutine to check for overflow
        add             $s0, $s0, $t4        # store product into s0, the destination vector D

#       Lower half of the vectors
        sll             $t4, $t1, 8          # shift t4 and t5 left then right to gain their
                                             # sixth elements
        sll             $t5, $t3, 8
        srl             $t4, $t4, 24
        srl             $t5, $t5, 24
        mult            $t4, $t5             # multiply elements
        mflo            $t4                  # store the result into t4
        jal             Check                # jump to subroutine to check for overflow
        add             $s1, $s1, $t4        # store product into s1, the destination vector D
        sll             $s1, $s1, 16         # shift the bytes four times to make room for the
                                             # next sum
        sll             $t4, $t1, 24         # shift t4 and t5 left then right to gain their
                                             # eighth elements
        sll             $t5, $t3, 24
        srl             $t4, $t4, 24
        srl             $t5, $t5, 24
        mult            $t4, $t5             # multiply elements
        mflo            $t4                  # store the result into t4
        jal             Check                # jump to subroutine to check for overflow
        add             $s1, $s1, $t4        # store product into s1, the destination vector D

#       Function code for exit
exit:   ori             $v0, $zero, 10
        syscall
```

```
#       Subroutine to check for overflow
Check:  slt             $a0, $t4, $a1         # check if t4 is greater than 0xFFFF causing
overflow
        bne             $a0, $zero, End       # exits subroutine if not
        add             $t4, $zero, $a1       # sets $t4 to 0xFFFF is there's overflow
End:    jr              $ra                   # return to main routine
```

1.5 Vec_msums

```
#*******************************************************************************
# File name:          vec_msums.asm
# Version:            1.0
# Date:               November 12, 2018
# Programmer:         Ellen Burger
#
# Description:        Code implementing the instruction 'vec_msums d, a, b, c' into MIPS
#                     architecture.A, B, and C are 8-byte vectors with 2-byte wide elements,
#                     and D has 4-byte wide elements. The upper and lower halves of each
#                     vector are split between two registers. The instruction multiplies the
#                     first and second elements from A and B then adds them together with the
#                     element in C, then stores it into the corresponding element in vector D.
#
# Register usage:     $t0 - contains the upper half of vector A
#                     $t1 - contains the lower half of vector A
#                     $t2 - contains the upper half of vector B
#                     $t3 - contains the lower half of vector B
#                     $t4 - contains the lower half of vector C
#                     $t5 - contains the upper half of vector C
#                     $t6 - contains vector A's two bytes that are being multiplied, handling
#                     the even-numbered element positions. It holds the final sum of the
#                     operation and stores into vector D
#                     $t7 - contains vector B's two bytes that are being multiplied, handling
#                     the even-numbered element positions. It also holds vector C's
#                     even-numbered elements
#                     $t8 - contains vector A's two bytes that are being multiplied, handling
#                     the odd-numbered element positions
#                     $t9 - contains vector B's two bytes that are being multiplied, handling
#                     The odd-numbered element positions
#                     $s0 - contains the upper half of vector D
#                     $s1 - contains the lower half of vector D
#                     $a0 - set to 0 if there is a sum overflow
#                     $a1 - # set to 0xFFFF to check for overflows
#                     $ra - holds the return address
#*******************************************************************************

# Main code segment
.text
.globl  main
```

```
main:   li          $t0, 0x230CF14D     # load the lower half of vector A into t0
        li          $t1, 0x5C7F191A     # load the upper half of vector A into t1
        li          $t2, 0xA30C5BFD     # load the lower half of vector B into t2
        li          $t3, 0xC5FFC9EE     # load the upper half of vector B into t3
        li          $t4, 0x609E19F7     # load the lower half of vector C into t8
        li          $t5, 0x45670766     # load the upper half of vector C into t9
        li          $a1, 0xFFFF         # set a1 to 0xFFFF

#       Upper half of the vectors
        srl         $t6, $t0, 24        # shift t6 and t7 right to gain their first
                                        # #elements
        srl         $t7, $t2, 24
        mult        $t6, $t7            # multiply elements
        mflo        $t6                 # store the result into t6

        sll         $t8, $t0, 8         # shift t8 and t9 left then right to gain their
                                        # second elements
        sll         $t9, $t2, 8
        srl         $t8, $t8, 24
        srl         $t9, $t9, 24
        mult        $t8, $t9            # add elements and store the result in t8
        mflo        $t8                 # store the result into t8

        add         $t6, $t6, $t8       # add the two products
        srl         $t7, $t4, 16        # shift t7 to gain C's first element
        add         $t6, $t6, $t7       # add the sum of the products to t7
        jal         Check               # jump to subroutine to check for overflow
        add         $s0, $s0, $t6       # store sum into s0, the destination vector D
        sll         $s0, $s0, 16        # shift the bytes four times to make room for the
                                        # next sum
        sll         $t6, $t0, 16        # shift t6 and t7 left then right to gain their
                                        #third elements
        sll         $t7, $t2, 16
        srl         $t6, $t6, 24
        srl         $t7, $t7, 24
        mult        $t6, $t7            # multiply elements
        mflo        $t6                 # store the result into t6

        sll         $t8, $t0, 24        # shift t8 and t9 left then right to gain their
                                        # fourth elements
        sll         $t9, $t2, 24
        srl         $t8, $t8, 24
        srl         $t9, $t9, 24
        mult        $t8, $t9            # add elements and store the result in t8
        mflo        $t8                 # store the result into t8

        add         $t6, $t6, $t8       # add the two products
        sll         $t7, $t4, 16        # shift t7 left then right to gain C's second
```

```
                                        # element
        srl         $t7, $t7, 16
        add         $t6, $t6, $t7       # add the sum of the products to t7
        jal         Check               # jump to subroutine to check for overflow
        add         $s0, $s0, $t6       # store sum into s0, the destination vector D

#       Lower half of the vectors
        srl         $t6, $t1, 24        # shift t6 and t7 right to gain their fifth
                                        # elements
        srl         $t7, $t3, 24
        mult        $t6, $t7            # multiply elements
        mflo        $t6                 # store the result into t6

        sll         $t8, $t1, 8         # shift t8 and t9 left then right to gain their
                                        # sixth elements
        sll         $t9, $t3, 8
        srl         $t8, $t8, 24
        srl         $t9, $t9, 24
        mult        $t8, $t9            # add elements and store the result in t8
        mflo        $t8                 # store the result into t8

        add         $t6, $t6, $t8       # add the two products
        srl         $t7, $t5, 16        # shift t7 to gain C's second element
        add         $t6, $t6, $t7       # add the sum of the products to t7
        jal         Check               # jump to subroutine to check for overflow
        add         $s1, $s1, $t6       # store sum into s1, the destination vector D
        sll         $s1, $s1, 16        # shift the bytes four times to make room for the
                                        # next sum

        sll         $t6, $t1, 16        # shift t6 and t7 left then right to gain their
                                        # seventh elements
        sll         $t7, $t3, 16
        srl         $t6, $t6, 24
        srl         $t7, $t7, 24
        mult        $t6, $t7            # multiply elements
        mflo        $t6                 # store the result into t6

        sll         $t8, $t1, 24        # shift t8 and t9 left then right to gain their
                                        # eighth elements
        sll         $t9, $t3, 24
        srl         $t8, $t8, 24
        srl         $t9, $t9, 24
        mult        $t8, $t9            # add elements and store the result in t8
        mflo        $t8                 # store the result into t8

        add         $t6, $t6, $t8       # add the two products
        sll         $t7, $t5, 16        # shift t7 left then right to gain C's fourth
element
```

```
        srl         $t7, $t7, 16
        add         $t6, $t6, $t7       # add the sum of the products to t7
        jal         Check               # jump to subroutine to check for overflow
        add         $s1, $s1, $t6       # store sum into s1, the destination vector D

#       Function code for exit
exit:   ori         $v0, $zero, 10
        syscall

#       Subroutine to check for overflow
Check:  slt         $a0, $t6, $a1       # check if t6 is greater than 0xFFFF causing
                                        # overflow
        bne         $a0, $zero, End     # exits subroutine if not
        add         $t6, $zero, $a1     # sets $t6 to 0xFFFF is there's overflow
End:    jr          $ra                 # return to main routine
```

1.6 Vec_splat

```
#*******************************************************************************
# File name:          vec_splat.asm
# Version:            1.0
# Date:               November 12, 2018
# Programmer:         Ellen Burger
#
# Description:        Code implementing the instruction 'vec_splat d, a, b' into MIPS
#                     architecture. D, A, and B are 8-byte vectors with 2-byte wide
#                     elements. The upper and lower halves of each vector are split
#                     between two registers. The instruction reads vector B and uses
#                     its value to select an element from vector A, which is then
#                     placed into each element of vector D.
#
# Register usage:     $t0 - contains the upper half of vector A
#                     $t1 - contains the lower half of vector A
#                     $t2 - contains the upper half of vector B
#                     $t3 - contains the lower half of vector B, the position of vector A to
#                     be splatted
#                     $t4 - contains the element of vector A being splatted
#                     $t5 - used as a loop counter
#                     $t6 - used as a loop counter
#                     $s0 - contains the upper half of vector D
#                     $s1 - contains the lower half of vector D
#                     $a0 - set to 0 if the vector B goes beyond the vector range, then used
#                     as part of loop counters
#                     $ra - holds the return address
#*******************************************************************************

# Main code segment
.text
.globl  main

main:   li          $t0, 0x230C124D       # load the lower half of vector A into t0
        li          $t1, 0x057F192A       # load the upper half of vector A into t1
        li          $t2, 0x00000000       # load the lower half of vector B into t2
        li          $t3, 0x00000005       # load the upper half of vector B into t3

        jal         Check                 # jump to subroutine to check for range
        addi        $a0, $zero, 0x8       # sets a0 to 0x8 to later calculate how
                                          # many times a loop must iterate
        slti        $t4, $t3, 0x4         # checks if the value in B is less than four,
```

```
                                        # branches to Hi if not
        addi        $t6, $t3, 0x1       # increments B's value for use in loops
        beq         $t4, $zero, Hi

#       If the vector element is in the lower half of A
        add         $t4, $t0, $zero     # sets t4 to vector A's lower values
Low1:   sll         $t4, $t4, 4         # shifts t4 left until the selected element is the
                                        # two most significant bits
        addi        $t5, $t5, 0x1
        bne         $t6, $t5, Low1

        sub         $t6, $a0, $t6       # calcuates how many times t4 needs to be shifted
                                        # right
        add         $t5, $zero, $zero   # resets t5's loop counter
Low2:   srl         $t4, $t4, 4         # shifts t4 right until the selected element is
                                        # the two least significant bits
        addi        $t5, $t5, 0x1
        bne         $t6, $t5, Low2
        addi        $a0, $zero, 0x4     # sets a0 to 0x4 to act as a loop end amount
        add         $t5, $zero, $zero   # resets t5's loop counter
        j           Splat1              # jumps to the splat section of the instruction

#       If the vector element is in the upper half of A
Hi:     subi        $t6, $t6, 0x4       # calculates how many times t4 needs to be shifted
left
        add         $t4, $t1, $zero     # sets t4 to vector A's upper values
Hi1:    sll         $t4, $t4, 4         # shifts t4 left until the selected element is the
                                        # two most significant bits
        addi        $t5, $t5, 0x1
        bne         $t6, $t5, Hi1

        sub         $t6, $a0, $t6       # calcuates how many times t4 needs to be shifted
                                        # right
        add         $t5, $zero, $zero   # resets t5's loop counter
Hi2:    srl         $t4, $t4, 4         # shifts t4 right until the selected element is
                                        # the two least significant bits
        addi        $t5, $t5, 0x1
        bne         $t6, $t5, Hi2
        addi        $a0, $zero, 0x4     # sets a0 to 0x4 to act as a loop end amount
        add         $t5, $zero, $zero   # resets t5's loop counter

#       Implementing the splat
Splat1: sll         $s0, $s0, 8         # shifts s0 left two bits, loops until each
                                        # element of D is the selected element
        add         $s0, $s0, $t4       # adds the element from A into D's lower half
        addi        $t5, $t5, 0x1
        bne         $a0, $t5, Splat1
        add         $t5, $zero, $zero   # resets t5's loop counter
```

```
Splat2: sll          $s1, $s1, 8          # shifts s0 left two bits, loops until each
                                          # element of D is the selected element
        add          $s1, $s1, $t4        # adds the element from A into D's upper half
        addi         $t5, $t5, 0x1
        bne          $a0, $t5, Splat2

#       Function code for exit
exit:   ori          $v0, $zero, 10
        syscall

#       Subroutine to check for overflow
Check:  slti         $a0, $t6, 0x7        # check if t4 is greater than 0x7 which is beyond
                                          # the range of the vectors
        bne          $a0, $zero, End      # exits the subroutine if not
        j            exit                 # cancels the instruction if so
End:    jr           $ra                  # return to main routine
```

1.7 Vec_mergel

```
#*******************************************************************************
# File name:          vec_mergel.asm
# Version:            1.0
# Date:               November 12, 2018
# Programmer:         Ellen Burger
#
# Description:        Code implementing the instruction 'vec_mergel d, a, b' into MIPS
#                     architecture. D, A, and B are 8-byte vectors with 2-byte wide elements.
#                     The upper and lower halves of each vector are split between two
#                     registers. The instruction places the second halves of vectors A and B
#                     into vector D alternating. A4 => D0, B4 = > D1, A5 => D2 and so on.
#
# Register usage:     $t0 - contains the upper half of vector A
#                     $t1 - contains the lower half of vector A
#                     $t2 - contains the upper half of vector B
#                     $t3 - contains the lower half of vector B
#                     $t4 - contains vector A's element that's being placed into D
#                     $t5 - contains vector B's element that's being placed into D
#                     $s0 - contains the upper half of vector D
#                     $s1 - contains the lower half of vector D
#*******************************************************************************

# Main code segment
.text
.globl  main

main:   li              $t0, 0x5AF0A501       # load the lower half of vector A into t0
        li              $t1, 0xAB0155C3       # load the upper half of vector A into t1
        li              $t2, 0xA50F5A23       # load the lower half of vector B into t2
        li              $t3, 0xCD23AA3C       # load the upper half of vector B into t3

#       Upper half of vector D
        srl             $t4, $t1, 24          # shift t4 and t5 right to gain their fifth
                                              # elements
        srl             $t5, $t3, 24
        add             $s0, $s0, $t4         # adds vector A's element to vector D
        sll             $s0, $s0, 8           # shift the bytes twice to make room for the next
                                              # element
        add             $s0, $s0, $t5         # adds vector B's element to vector D
        sll             $s0, $s0, 8           # shift the bytes twice to make room for the next
```

```
                                        # element

        sll         $t4, $t1, 8         # shift t4 and t5 left then right to gain their
                                        # sixth elements
        sll         $t5, $t3, 8
        srl         $t4, $t4, 24
        srl         $t5, $t5, 24
        add         $s0, $s0, $t4       # adds vector A's element to vector D
        sll         $s0, $s0, 8         # shift the bytes twice to make room for the next
                                        # element
        add         $s0, $s0, $t5       # adds vector B's element to vector D

#       Lower half of vector D
        sll         $t4, $t1, 16        # shift t4 and t5 left then right to gain their
                                        # seventh elements
        sll         $t5, $t3, 16
        srl         $t4, $t4, 24
        srl         $t5, $t5, 24
        add         $s1, $s1, $t4       # adds vector A's element to vector D
        sll         $s1, $s1, 8         # shift the bytes twice to make room for the next
                                        # element
        add         $s1, $s1, $t5       # adds vector B's element to vector D
        sll         $s1, $s1, 8         # shift the bytes twice to make room for the next
                                        # element

        sll         $t4, $t1, 24        # shift t4 and t5 left then right to gain their
                                        # eighth elements
        sll         $t5, $t3, 24
        srl         $t4, $t4, 24
        srl         $t5, $t5, 24
        add         $s1, $s1, $t4       # adds vector A's element to vector D
        sll         $s1, $s1, 8         # shift the bytes twice to make room for the next
                                        # element
        add         $s1, $s1, $t5       # adds vector B's element to vector D

#       Function code for exit
exit:   ori         $v0, $zero, 10
        syscall
```

1.8 Vec_mergeh

```
#*******************************************************************************
# File name:          vec_mergeh.asm
# Version:            1.0
# Date:               November 12, 2018
# Programmer:         Ellen Burger
#
# Description:        Code implementing the instruction 'vec_mergeh d, a, b' into MIPS
#                     architecture. D, A, and B are 8-byte vectors with 2-byte wide elements.
#                     The upper and lower halves of each vector are split between two
#                     registers. The instruction places the first halves of vectors A and B
#                     into vector D alternating. A0 => D0, B0 = > D1, A1 => D2 and so on.
#
# Register usage:     $t0 - contains the upper half of vector A
#                     $t1 - contains the lower half of vector A
#                     $t2 - contains the upper half of vector B
#                     $t3 - contains the lower half of vector B
#                     $t4 - contains vector A's element that's being placed into D
#                     $t5 - contains vector B's element that's being placed into D
#                     $s0 - contains the upper half of vector D
#                     $s1 - contains the lower half of vector D
#*******************************************************************************

# Main code segment
.text
.globl  main

main:   li              $t0, 0x5AF0A501       # load the lower half of vector A into t0
        li              $t1, 0xAB0155C3       # load the upper half of vector A into t1
        li              $t2, 0xA50F5A23       # load the lower half of vector B into t2
        li              $t3, 0xCD23AA3C       # load the upper half of vector B into t3

#       Upper half of vector D
        srl             $t4, $t0, 24          # shift t4 and t5 right to gain their first
                                              # elements
        srl             $t5, $t2, 24
        add             $s0, $s0, $t4         # adds vector A's element to vector D
        sll             $s0, $s0, 8           # shift the bytes twice to make room for the next
                                              # element
        add             $s0, $s0, $t5         # adds vector B's element to vector D
        sll             $s0, $s0, 8           # shift the bytes twice to make room for the next
```

```
                                        # element

        sll         $t4, $t0, 8         # shift t4 and t5 left then right to gain their
                                        # second elements
        sll         $t5, $t2, 8
        srl         $t4, $t4, 24
        srl         $t5, $t5, 24
        add         $s0, $s0, $t4       # adds vector A's element to vector D
        sll         $s0, $s0, 8         # shift the bytes twice to make room for the next
                                        # element
        add         $s0, $s0, $t5       # adds vector B's element to vector D

#       Lower half of vector D
        sll         $t4, $t0, 16        # shift t4 and t5 left then right to gain their
                                        # third elements
        sll         $t5, $t2, 16
        srl         $t4, $t4, 24
        srl         $t5, $t5, 24
        add         $s1, $s1, $t4       # adds vector A's element to vector D
        sll         $s1, $s1, 8         # shift the bytes twice to make room for the next
                                        # element
        add         $s1, $s1, $t5       # adds vector B's element to vector D
        sll         $s1, $s1, 8         # shift the bytes twice to make room for the next
                                        # element

        sll         $t4, $t0, 24        # shift t4 and t5 left then right to gain their
                                        # fourth elements
        sll         $t5, $t2, 24
        srl         $t4, $t4, 24
        srl         $t5, $t5, 24
        add         $s1, $s1, $t4       # adds vector A's element to vector D
        sll         $s1, $s1, 8         # shift the bytes twice to make room for the next
                                        # element
        add         $s1, $s1, $t5       # adds vector B's element to vector D

#       Function code for exit
exit:   ori         $v0, $zero, 10
        syscall
```

1.9 Vec_pack

```
#*******************************************************************************
# File name:          vec_pack.asm
# Version:            1.0
# Date:               November 12, 2018
# Programmer:         Ellen Burger
#
# Description:        Code implementing the instruction 'vec_pack d, a, b' into MIPS
#                     architecture. D, A, and B are 8-byte vectors with 2-byte wide elements.
#                     The upper and lower halves of each vector are split between two
#                     registers. The instruction combines the elements of A and B such that
#                     each element of D is a truncation of the lower half of two consecutive #
               elements in A, with the lower half of D taken from B. The first two
#                     elements of A are combined and placed into D's first element, the next
#                     two into D's second element, and so on. D's fifth element is when it
#                     begins using B's elements.
#
# Register usage:     $t0 - contains the upper half of vector A
#                     $t1 - contains the lower half of vector A
#                     $t2 - contains the upper half of vector B
#                     $t3 - contains the lower half of vector B
#                     $t4 - contains the lower bit of vector A's even-numbered elements, then
#                     receives the sum with the next element's bit and stored into vector D.
#                     This is repeated for B.
#                     $t5 - contains the lower bit of vector A's odd-numbered elements. This
#                     is repeated for B.
#                     $s0 - contains the upper half of vector D
#                     $s1 - contains the lower half of vector D
#*******************************************************************************

# Main code segment
.text
.globl  main

main:   li          $t0, 0x5AFB6C1D       # load the lower half of vector A into t0
        li          $t1, 0xAE5FC041       # load the upper half of vector A into t1
        li          $t2, 0x52F3A415       # load the lower half of vector B into t2
        li          $t3, 0xA657C849       # load the upper half of vector B into t3
```

```
#       Vector A being packed
        sll             $t4, $t0, 4          # shift t4 and t5 left and right to gain the lower
                                             # half of the first and second elements of A
        sll             $t5, $t0, 12
        srl             $t4, $t4, 28
        sll             $t4, $t4, 4
        srl             $t5, $t5, 28
        add             $t4, $t4, $t5        # add elements and store the result in t4
        add             $s0, $s0, $t4        # store sum into s0, the destination vector D
        sll             $s0, $s0, 8          # shift the bytes twice to make room for the next
                                             # sum

        sll             $t4, $t0, 20         # shift t4 and t5 left and right to gain the lower
                                             # half of the third and fourth elements of A
        sll             $t5, $t0, 28
        srl             $t4, $t4, 28
        sll             $t4, $t4, 4
        srl             $t5, $t5, 28
        add             $t4, $t4, $t5        # add elements and store the result in t4
        add             $s0, $s0, $t4        # store sum into s0, the destination vector D
        sll             $s0, $s0, 8          # shift the bytes twice to make room for the next
                                             # sum

        sll             $t4, $t1, 4          # shift t4 and t5 left and right to gain the lower
                                             # half of the fifth and sixth elements of A
        sll             $t5, $t1, 12
        srl             $t4, $t4, 28
        sll             $t4, $t4, 4
        srl             $t5, $t5, 28
        add             $t4, $t4, $t5        # add elements and store the result in t4
        add             $s0, $s0, $t4        # store sum into s0, the destination vector D
        sll             $s0, $s0, 8          # shift the bytes twice to make room for the next
                                             # sum

        sll             $t4, $t1, 20         # shift t4 and t5 left and right to gain the lower
                                             # half of the seventh and eighth elements of A
        sll             $t5, $t1, 28
        srl             $t4, $t4, 28
        sll             $t4, $t4, 4
        srl             $t5, $t5, 28
        add             $t4, $t4, $t5        # add elements and store the result in t4
        add             $s0, $s0, $t4        # store sum into s0, the destination vector D

#       Vector B being packed
        sll             $t4, $t2, 4          # shift t4 and t5 left and right to gain the lower
                                             # half of the first and second elements of B
        sll             $t5, $t2, 12
        srl             $t4, $t4, 28
```

```
        sll         $t4, $t4, 4
        srl         $t5, $t5, 28
        add         $t4, $t4, $t5       # add elements and store the result in t4
        add         $s1, $s1, $t4       # store sum into s1, the destination vector D
        sll         $s1, $s1, 8         # shift the bytes twice to make room for the next
                                        # sum

        sll         $t4, $t2, 20        # shift t4 and t5 left and right to gain the lower
                                        # half of the third and fourth elements of B
        sll         $t5, $t2, 28
        srl         $t4, $t4, 28
        sll         $t4, $t4, 4
        srl         $t5, $t5, 28
        add         $t4, $t4, $t5       # add elements and store the result in t4
        add         $s1, $s1, $t4       # store sum into s1, the destination vector D
        sll         $s1, $s1, 8         # shift the bytes twice to make room for the next
                                        # sum

        sll         $t4, $t3, 4         # shift t4 and t5 left and right to gain the lower
                                        # half of the fifth and sixth elements of B
        sll         $t5, $t3, 12
        srl         $t4, $t4, 28
        sll         $t4, $t4, 4
        srl         $t5, $t5, 28
        add         $t4, $t4, $t5       # add elements and store the result in t4
        add         $s1, $s1, $t4       # store sum into s1, the destination vector D
        sll         $s1, $s1, 8         # shift the bytes twice to make room for the next
                                        # sum

        sll         $t4, $t3, 20        # shift t4 and t5 left and right to gain the lower
                                        # half of the seventh and eighth elements of B
        sll         $t5, $t3, 28
        srl         $t4, $t4, 28
        sll         $t4, $t4, 4
        srl         $t5, $t5, 28
        add         $t4, $t4, $t5       # add elements and store the result in t4
        add         $s1, $s1, $t4       # store sum into s1, the destination vector D

#       Function code for exit
exit:   ori         $v0, $zero, 10
        syscall
```

1.10 Vec_perm

```
#*******************************************************************************
# File name:          vec_perm.asm
# Version:            1.0
# Date:               November 12, 2018
# Programmer:         Ellen Burger
#
# Description:        Code implementing the instruction 'vec_perm d, a, b, c' into
#                     MIPS architecture. D, A, B, and C are 8-byte vectors with 2-byte wide
#                     elements. The upper and lower halves of each vector are split between
#                     two registers. The instruction reads vector C's elements one by one.
#                     The upper half decides whether to use vector A or vector B, and the
#                     lower half decides which element in the vector to use. The selected
#                     element is stored into D's element that corresponds with C's element,
#                     and is repeated for all eight.
#
# Register usage:     $t0 - contains the upper half of vector A
#                     $t1 - contains the lower half of vector A
#                     $t2 - contains the upper half of vector B
#                     $t3 - contains the lower half of vector B
#                     $t4 - contains the upper half of vector C
#                     $t5 - contains the lower half of vector C
#                     $t6 - contains the element from C to be used in the permutation
#                     $t9 - contains the constant 0x4
#                     $s0 - contains the upper half of vector D
#                     $s1 - contains the lower half of vector D
#                     $a0 - contains the element specifier from C, then the element selected
#                     from vector A or B and stores into D
#                     $a1 - contains the position of the selected element and acts as a loop
#                     counter
#                     $a2 - acts as a loop counter incremented by 1 each iteration
#                     $a3 - contains 0 or 1, specifying if the element is to be stored into
#                     D's upper or lower half
#                     $ra - holds the return address
#*******************************************************************************

# Main code segment
.text
.globl  main

main:   li          $t0, 0xA567013D       # load the upper half of vector A into t0
        li          $t1, 0xAB45393C       # load the lower half of vector A into t1
```

```
li          $t2, 0xEFC54D23      # load the upper half of vector B into t2
li          $t3, 0x1277AACD      # load the lower half of vector B into t3
li          $t4, 0x04171002      # load the upper half of vector C into t4
li          $t5, 0x13050105      # load the lower half of vector C into t5
li          $t9, 0x4             # sets t9 to constant 0x4

srl         $t6, $t4, 24         # sets t6 to C's first element
jal         Perm
sll         $s0, $s0, 8          # shifts s0 left two bits to make room for the
                                 # next element

sll         $t6, $t4, 8          # sets t6 to C's second element
srl         $t6, $t6, 24
jal         Perm
sll         $s0, $s0, 8          # shifts s0 left two bits to make room for the
                                 # next element

sll         $t6, $t4, 16         # sets t6 to C's third element
srl         $t6, $t6, 24
jal         Perm
sll         $s0, $s0, 8          # shifts s0 left two bits to make room for the
                                 # next element

sll         $t6, $t4, 24         # sets t6 to C's fourth element
srl         $t6, $t6, 24
jal         Perm
addi        $a3, $a3, 0x1

srl         $t6, $t5, 24         # sets t6 to C's fifth element
jal         Perm
sll         $s1, $s1, 8          # shifts s1 left two bits to make room for the
                                 # next element

sll         $t6, $t5, 8          # sets t6 to C's sixth element
srl         $t6, $t6, 24
jal         Perm
sll         $s1, $s1, 8          # shifts s1 left two bits to make room for the
                                 # next element

sll         $t6, $t5, 16         # sets t6 to C's seventh element
srl         $t6, $t6, 24
jal         Perm
sll         $s1, $s1, 8          # shifts s1 left two bits to make room for the
                                 # next element

sll         $t6, $t5, 24         # sets t6 to C's eigth element
srl         $t6, $t6, 24
jal         Perm
```

```
#       Function code for exit
exit:   ori             $v0, $zero, 10
        syscall

#       Subroutine implementing permutation
Perm:   srl             $a0, $t6, 4           # sets a0 to the element specifier
        sll             $a1, $t6, 28          # sets a1 to the position of the element + 1
        srl             $a1, $a1, 28
        addi            $a1, $a1, 0x1
        bne             $a0, $zero, VecB      # branches to VecB if specifier = 1

VecA:   slti            $a0, $a1, 0x5         # branches to UpA if position is in the first half
                                              # of vector A
        bne             $a0, $zero, UpA
        add             $a0, $t1, $zero       # sets a0 to the lower half of vector A
        subi            $a1, $a1, 0x4         # subtracts 4 from the position to act as a loop
                                              # counter
        j               P1                    # jumps to the first segment of permutation
                                              # instructions
UpA:    add             $a0, $t0, $zero       # sets a0 to the upper half of vector A

P1:     addi            $a2, $a2, 0x1         # increments a2 by 1 every loop
        beq             $a1, $a2, P2          # branches to second segment when a2 = a1, the
                                              # element position
        sll             $a0, $a0, 8           # shifts vector A left until the selected element
                                              # is in the most significant bits
        j               P1

P2:     addi            $a1, $zero, 0x3       # sets a1 to 3 to act as a loop counter
        add             $a2, $zero, $zero     # resets a2's loop counter

P3:     srl             $a0, $a0, 8           # shifts a0 right until the selected element is in
                                              # the least significant bits
        addi            $a2, $a2, 0x1         # increments a2 by 1 every loop
        bne             $a1, $a2, P3          # branches to start of segment when a2 != a1, the
                                              # element position
        add             $a2, $zero, $zero     # sets a2 to zero

        bne             $a3, $zero, P4        # branches to fourth segment if a4 != zero
        add             $s0, $s0, $a0         # adds the element to vector D's upper half
        j               ExitA
P4:     add             $s1, $s1, $a0         # adds the element to vector D's lower half

ExitA:  jr              $ra                   # jumps back to main section

VecB:    slti           $a0, $a1, 0x5         # branches to UpA if position is in the first half
```

```
                                        # of vector B
        bne         $a0, $zero, UpB
        add         $a0, $t3, $zero     # sets a0 to the lower half of vector B
        subi        $a1, $a1, 0x4       # subtracts 4 from the position to act as a loop
                                        # counter
        j           P1                  # jumps to the first segment of permutation
                                        # instructions
UpB:    add         $a0, $t2, $zero     # sets a0 to the upper half of vector B
        j           P1
```

1.11 Vec_cmpeq

```
#*******************************************************************************
# File name:          vec_cmpeq.asm
# Version:            1.0
# Date:               November 12, 2018
# Programmer:         Ellen Burger
#
# Description:        Code implementing the instruction 'vec_cmpeq d, a, b' into MIPS
#                     architecture. D, A, and B are 8-byte vectors with 2-byte wide elements.
#                     The upper and lower halves of each vector are split between two
#                     registers. The instruction checks if the element of A equals B's, and
#                     if so, saves 0xFF into D's corresponding element. If not, remains 0x0.
#                     This repeats for all eight elements.
#
# Register usage:     $t0 - contains the upper half of vector A
#                     $t1 - contains the lower half of vector A
#                     $t2 - contains the upper half of vector B
#                     $t3 - contains the lower half of vector B
#                     $t4 - contains vector A's two bytes that are being compared
#                     $t5 - contains vector B's two bytes that are being compared
#                     $s0 - contains the upper half of vector D
#                     $s1 - contains the lower half of vector D
#                     $ra - holds the return address
#*******************************************************************************

# Main code segment
.text
.globl  main

main:   li              $t0, 0x5AFB6C1D       # load the lower half of vector A into t0
        li              $t1, 0xA65FC040       # load the upper half of vector A into t1
        li              $t2, 0x52FBA415       # load the lower half of vector B into t2
        li              $t3, 0xAE5FC841       # load the upper half of vector B into t3

#       Upper half of the vectors
        srl             $t4, $t0, 24          # shift t4 and t5 right to gain their first
                                              # elements
        srl             $t5, $t2, 24
        jal             EquHi                 # jump to subroutine to check if A equals B
        sll             $s0, $s0, 8           # shift the bytes twice to make room for the next
```

```
                                        # sum

        sll         $t4, $t0, 8         # shift t4 and t5 left then right to gain their
                                        # second elements
        sll         $t5, $t2, 8
        srl         $t4, $t4, 24
        srl         $t5, $t5, 24
        jal         EquHi               # jump to subroutine to check if A is equals B
        sll         $s0, $s0, 8         # shift the bytes twice to make room for the next
                                        # sum

        sll         $t4, $t0, 16        # shift t4 and t5 left then right to gain their
                                        # third elements
        sll         $t5, $t2, 16
        srl         $t4, $t4, 24
        srl         $t5, $t5, 24
        jal         EquHi               # jump to subroutine to check if A is equals B
        sll         $s0, $s0, 8         # shift the bytes twice to make room for the next
                                        # sum

        sll         $t4, $t0, 24        # shift t4 and t5 left then right to gain their
                                        # fourth elements
        sll         $t5, $t2, 24
        srl         $t4, $t4, 24
        srl         $t5, $t5, 24
        jal         EquHi               # jump to subroutine to check if A is equals B

#       Lower half of the vectors
        srl         $t4, $t1, 24        # shift t4 and t5 right to gain their fifth
                                        # elements
        srl         $t5, $t3, 24
        jal         EquLo               # jump to subroutine to check if A is equals B
        sll         $s1, $s1, 8         # shift the bytes twice to make room for the next
                                        # sum

        sll         $t4, $t1, 8         # shift t4 and t5 left then right to gain their
                                        # sixth elements
        sll         $t5, $t3, 8
        srl         $t4, $t4, 24
        srl         $t5, $t5, 24
        jal         EquLo               # jump to subroutine to check if A is equals B
        sll         $s1, $s1, 8         # shift the bytes twice to make room for the next
                                        # sum


        sll         $t4, $t1, 16        # shift t4 and t5 left then right to gain their
                                        # seventh elements
        sll         $t5, $t3, 16
```

```
        srl         $t4, $t4, 24
        srl         $t5, $t5, 24
        jal         EquLo               # jump to subroutine to check if A is equals B
        sll         $s1, $s1, 8         # shift the bytes twice to make room for the next
                                        # sum


        sll         $t4, $t1, 24        # shift t4 and t5 left then right to gain their
                                        # eighth elements
        sll         $t5, $t3, 24
        srl         $t4, $t4, 24
        srl         $t5, $t5, 24
        jal         EquLo               # jump to subroutine to check if A is equals B

#       Function code for exit
exit:   ori         $v0, $zero, 10
        syscall

#       Subroutine to check for overflow
EquHi:  bne         $t4, $t5, EndH      # exits subroutine if A != B
        addi        $s0, $s0, 0xFF      # sets D's element to 0xFF if A = B
EndH:   jr          $ra                 # return to main routine

EquLo:  bne         $t4, $t5, EndL      # exits subroutine if A != B
        addi        $s1, $s1, 0xFF      # sets D's element to 0xFF if A = B
EndL:   jr          $ra                 # return to main routine
```

1.12 Vec_cmpltu

```
#*******************************************************************************
# File name:          vec_cmpltu.asm
# Version:            1.0
# Date:               November 12, 2018
# Programmer:         Ellen Burger
#
# Description:        Code implementing the instruction 'vec_cmpltu d, a, b' into MIPS
#                     architecture. D, A, and B are 8-byte vectors with 2-byte wide elements.
#                     The upper and lower halves of each vector are split between two
#                     registers. The instruction checks if the element of A is less than B's,
#                     and if so, saves 0xFF into D's corresponding element. If not, remains
#                     0x0. This repeats for all eight elements.
#
# Register usage:     $t0 - contains the upper half of vector A
#                     $t1 - contains the lower half of vector A
#                     $t2 - contains the upper half of vector B
#                     $t3 - contains the lower half of vector B
#                     $t4 - contains vector A's two bytes that are being compared. Receives 1
#                     if it is less than B's, 0 otherwise
#                     $t5 - contains vector B's two bytes that are being compared
#                     $s0 - contains the upper half of vector D
#                     $s1 - contains the lower half of vector D
#                     $ra - holds the return address
#*******************************************************************************

# Main code segment
.text
.globl  main

main:   li          $t0, 0x5AFB6C1D       # load the lower half of vector A into t0
        li          $t1, 0xA65FC040       # load the upper half of vector A into t1
        li          $t2, 0x52FBA415       # load the lower half of vector B into t2
        li          $t3, 0xAE5FC841       # load the upper half of vector B into t3

#       Upper half of the vectors
        srl         $t4, $t0, 24          # shift t4 and t5 right to gain their first
                                          # elements
        srl         $t5, $t2, 24
        slt         $t4, $t4, $t5         # compares the elements
        jal         CmpHi                 # jump to subroutine to check if A is less than B
        sll         $s0, $s0, 8           # shift the bytes twice to make room for the next
```

```
                                        # sum


        sll         $t4, $t0, 8         # shift t4 and t5 left then right to gain their
                                        # second elements
        sll         $t5, $t2, 8
        srl         $t4, $t4, 24
        srl         $t5, $t5, 24
        slt         $t4, $t4, $t5       # compares the elements
        jal         CmpHi               # jump to subroutine to check if A is less than B
        sll         $s0, $s0, 8         # shift the bytes twice to make room for the next
                                        # sum

        sll         $t4, $t0, 16        # shift t4 and t5 left then right to gain their
                                        # third elements
        sll         $t5, $t2, 16
        srl         $t4, $t4, 24
        srl         $t5, $t5, 24
        slt         $t4, $t4, $t5       # compares the elements
        jal         CmpHi               # jump to subroutine to check if A is less than B
        sll         $s0, $s0, 8         # shift the bytes twice to make room for the next
                                        # sum

        sll         $t4, $t0, 24        # shift t4 and t5 left then right to gain their
                                        # fourth elements
        sll         $t5, $t2, 24
        srl         $t4, $t4, 24
        srl         $t5, $t5, 24
        slt         $t4, $t4, $t5       # compares the elements
        jal         CmpHi               # jump to subroutine to check if A is less than B

#       Lower half of the vectors
        srl         $t4, $t1, 24        # shift t4 and t5 right to gain their fifth
                                        # elements
        srl         $t5, $t3, 24
        slt         $t4, $t4, $t5       # compares the elements
        jal         CmpLow              # jump to subroutine to check if A is less than B
        sll         $s1, $s1, 8         # shift the bytes twice to make room for the next
                                        # sum

        sll         $t4, $t1, 8         # shift t4 and t5 left then right to gain their
                                        # sixth elements
        sll         $t5, $t3, 8
        srl         $t4, $t4, 24
        srl         $t5, $t5, 24
        slt         $t4, $t4, $t5       # compares the elements
        jal         CmpLow              # jump to subroutine to check if A is less than B
        sll         $s1, $s1, 8         # shift the bytes twice to make room for the next
```

```
                                        # sum

        sll         $t4, $t1, 16        # shift t4 and t5 left then right to gain their
                                        # seventh elements
        sll         $t5, $t3, 16
        srl         $t4, $t4, 24
        srl         $t5, $t5, 24
        slt         $t4, $t4, $t5       # compares the elements
        jal         CmpLow              # jump to subroutine to check if A is less than B
        sll         $s1, $s1, 8         # shift the bytes twice to make room for the next
                                        # sum

        sll         $t4, $t1, 24        # shift t4 and t5 left then right to gain their
                                        # eighth elements
        sll         $t5, $t3, 24
        srl         $t4, $t4, 24
        srl         $t5, $t5, 24
        slt         $t4, $t4, $t5       # compares the elements
        jal         CmpLow              # jump to subroutine to check if A is less than B

#       Function code for exit
exit:   ori         $v0, $zero, 10
        syscall

#       Subroutine to check for overflow
CmpHi:  beq         $t4, $zero, EndH    # exits subroutine if A is less than B
        addi        $s0, $s0, 0xFF      # sets D's element to 0xFF if not
EndH:   jr          $ra                 # return to main routine

CmpLow: beq         $t4, $zero, EndL    # exits subroutine if A is less than B
        addi        $s1, $s1, 0xFF      # sets D's element to 0xFF if not
EndL:   jr          $ra                 # return to main routine
```

2. Application Specific Enhancements

2.1 Vec_and

```
#*******************************************************************************
# File name:          vec_and.asm
# Version:            1.0
# Date:               November 12, 2018
# Programmer:         Ellen Burger
#
# Description:        Code implementing the instruction 'vec_and d, a, b' into MIPS
#                     architecture. D, A, and B are 8-byte vectors with 2-byte wide elements.
#                     The upper and lower halves of each vector are split between two
#                     registers. The instruction and's each element from A and B one at a
#                     time and stores it into the corresponding element in vector D.
#
# Register usage:     $t0 - contains the upper half of vector A
#                     $t1 - contains the lower half of vector A
#                     $t2 - contains the upper half of vector B
#                     $t3 - contains the lower half of vector B
#                     $t4 - contains vector A's two bytes that are being and'd, then receives
#                     the result to be stored into vector D. Set to 0xff if the result causes
#                     overflow
#                     $t5 - contains vector B's two bytes that are being and'd
#                     $s0 - contains the upper half of vector D
#                     $s1 - contains the lower half of vector D
#*******************************************************************************

# Main code segment
.text
.globl  main

main:   li              $t0, 0x233C475D       # load the upper half of vector A into t0
        li              $t1, 0x087F196F       # load the lower half of vector A into t1
        li              $t2, 0x981963C5       # load the upper half of vector B into t2
        li              $t3, 0x5E80B36E       # load the lower half of vector B into t3

#       Upper half of the vectors
        srl             $t4, $t0, 24          # shift t4 and t5 right to gain their first
                                              # elements
        srl             $t5, $t2, 24
```

```
        and         $t4, $t4, $t5       # and elements and store the result in t4
        add         $s0, $t4, $zero     # store result into s0, the destination vector D
        sll         $s0, $s0, 8         # shift the bytes twice to make room for the next
                                        # result

        sll         $t4, $t0, 8         # shift t4 and t5 left then right to gain their
                                        # second elements
        sll         $t5, $t2, 8
        srl         $t4, $t4, 24
        srl         $t5, $t5, 24
        and         $t4, $t4, $t5       # and elements and store the result in t4
        add         $s0, $s0, $t4       # store result into s0, the destination vector D
        sll         $s0, $s0, 8         # shift the bytes twice to make room for the next
                                        # result

        sll         $t4, $t0, 16        # shift t4 and t5 left then right to gain their
                                        # third elements
        sll         $t5, $t2, 16
        srl         $t4, $t4, 24
        srl         $t5, $t5, 24
        and         $t4, $t4, $t5       # and elements and store the result in t4
        add         $s0, $s0, $t4       # store result into s0, the destination vector D
        sll         $s0, $s0, 8         # shift the bytes twice to make room for the next
                                        # result

        sll         $t4, $t0, 24        # shift t4 and t5 left then right to gain their
                                        # fourth elements
        sll         $t5, $t2, 24
        srl         $t4, $t4, 24
        srl         $t5, $t5, 24
        or          $t4, $t4, $t5       # and elements and store the result in t4
        add         $s0, $s0, $t4       # store result into s0, the destination vector D

#       Lower half of the vectors
        srl         $t4, $t1, 24        # shift t4 and t5 right to gain their fifth
                                        # elements
        srl         $t5, $t3, 24
        and         $t4, $t4, $t5       # and elements and store the result in t4
        add         $s1, $t4, $zero     # store result into s1, the destination vector D
        sll         $s1, $s1, 8         # shift the bytes twice to make room for the next
                                        # result

        sll         $t4, $t1, 8         # shift t4 and t5 left then right to gain their
                                        # sixth elements
        sll         $t5, $t3, 8
        srl         $t4, $t4, 24
        srl         $t5, $t5, 24
        and         $t4, $t4, $t5       # and elements and store the result in t4
```

```
        add         $s1, $s1, $t4       # store result into s1, the destination vector D
        sll         $s1, $s1, 8         # shift the bytes twice to make room for the next
                                        # result

        sll         $t4, $t1, 16        # shift t4 and t5 left then right to gain their
                                        # seventh elements
        sll         $t5, $t3, 16
        srl         $t4, $t4, 24
        srl         $t5, $t5, 24
        and         $t4, $t4, $t5       # and elements and store the result in t4
        add         $s1, $s1, $t4       # store result into s1, the destination vector D
        sll         $s1, $s1, 8         # shift the bytes twice to make room for the next
                                        # result

        sll         $t4, $t1, 24        # shift t4 and t5 left then right to gain their
                                        # eighth elements
        sll         $t5, $t3, 24
        srl         $t4, $t4, 24
        srl         $t5, $t5, 24
        and         $t4, $t4, $t5       # and elements and store the result in t4
        add         $s1, $s1, $t4       # store result into s1, the destination vector D

#       Function code for exit
exit:   ori         $v0, $zero, 10
        syscall
```

2.2 Vec_beq

```
#*******************************************************************************
# File name:          vec_beq.asm
# Version:            1.0
# Date:               November 12, 2018
# Programmer:         Ellen Burger
#
# Description:        Code implementing the instruction 'vec_beq a, b, imm' into MIPS
#                     architecture. A and B are 8-byte vectors with 2-byte wide elements. The
#                     upper and lower halves of each vector are split between two registers.
#                     The instruction checks if the vectors are equal element by element. If
#                     so, branches to the address in imm which is represented by "Target" in
#                     this example. Moves onto the next instruction as normal if not.
#
# Register usage:     $t0 - contains the upper half of vector A
#                     $t1 - contains the lower half of vector A
#                     $t2 - contains the upper half of vector B
#                     $t3 - contains the lower half of vector B
#*******************************************************************************

# Main code segment
.text
.globl  main

main:   li          $t0, 0x233C475D       # load the lower half of vector A into t0
        li          $t1, 0x087F196F       # load the upper half of vector A into t1
        li          $t2, 0x233C475D       # load the lower half of vector B into t2
        li          $t3, 0x087F196F       # load the upper half of vector B into t3

        beq         $t0, $t2, LoTest      # checks if the upper halves of A and B are
                                          # equal and branches to Lo. Test if so. Exits if
                                          # not
        j           exit

LoTest: beq         $t1, $t3, Target      # checks if the lower halves of A and B are
                                          # equal and branches to Target if so. Exits if not

#       Function code for exit
exit:   ori         $v0, $zero, 10
        syscall
```

```
#       Branch to the target
Target: j             exit                  # Instruction branches to Target if A = B.
                                            # In this example, Target then exits the
                                            # instruction
```

2.3 Vec_bne

```
#*******************************************************************************
# File name:          vec_bne.asm
# Version:            1.0
# Date:               November 12, 2018
# Programmer:         Ellen Burger
#
# Description:        Code implementing the instruction 'vec_beq a, b, imm' into MIPS
#                     architecture. A and B are 8-byte vectors with 2-byte wide elements. The
#                     upper and lower halves of each vector are split between two registers.
#                     The instruction checks if the vectors are inqeual at any element. If
#                     so, branches to the address in imm which is represented by "Target" in
#                     this example. Moves onto the next instruction as normal if not.
#
# Register usage:     $t0 - contains the upper half of vector A
#                     $t1 - contains the lower half of vector A
#                     $t2 - contains the upper half of vector B
#                     $t3 - contains the lower half of vector B
#*******************************************************************************

# Main code segment
.text
.globl  main

main:   li          $t0, 0x233C475D             # load the lower half of vector A into t0
        li          $t1, 0x087F196F             # load the upper half of vector A into t1
        li          $t2, 0x233C475D             # load the lower half of vector B into t2
        li          $t3, 0x087F087F             # load the upper half of vector B into t3

        bne         $t0, $t2, Target            # checks if the upper halves of A and B
                                                # aren't equal and branches to Target if
                                                # so. Continues to check the lower halves
                                                # if not
        bne         $t1, $t3, Target            # checks if the lower halves of A and B
                                                # aren't equal and branches to Target if
                                                # so. Exits if not

#       Function code for exit
exit:   ori         $v0, $zero, 10
        syscall
```

```
#      Branch to the immediate value
Target: j             exit                        # Instruction branches to Target if A != B.
                                                  # In tHIS example, Target then exits the
                                                  # instruction
```

2.4 Vec_decry

```
#*******************************************************************************
# File name:          vec_decry.asm
# Version:            1.0
# Date:               November 12, 2018
# Programmer:         Ellen Burger
#
# Description:        Code implementing the instruction 'vec_decry d, a, b' into MIPS
#                     architecture. D and B are 8-byte vectors; D has two-byte wide elements
#                     and B has four-byte wide elements. A is a 16-byte vector with four-byte
#                     wide elements. The upper and lower halves of D and B are split between
#                     two registers, and A quartered between four. The instruction decrypts
#                     each element from A using RSA decryption algorithm. The algorithm's p,
#                     q, e, and d values are supplied by each of B's elements. The decrypted
#                     elements of A are stored into D's elements one at a time.
#
# Register usage:     $t0 - contains the first quarter of vector A
#                     $t1 - contains the second quarter of vector A
#                     $t2 - contains the third quarter of vector A
#                     $t3 - contains the fourth quarter of vector A
#                     $t4 - contains the upper half of vector B, p in the upper element and q
#                     in the lower
#                     $t5 - contains the lower half of vector B, e in the upper element and d
#                     in the lower
#                     $t6 - contains n which is found by p * q
#                     $t7 - contains d
#                     $t8 - contains the element from A to be decrypted, then stores into D
#                     after decryption
#                     $s0 - contains the upper half of vector D
#                     $s1 - contains the lower half of vector D
#                     $a0 - acts as a loop counter, ending the RSA algorithm when equaling d
#                     $a1 - used to calculate the decrypted value in the RSA algorithm
#                     $ra - holds the return address
#*******************************************************************************

# Main code segment
.text
.globl  main

main:   li              $t0, 0x024d0399       # load the first quarter of vector A into t0
        li              $t1, 0x029d0449       # load the second quarter of vector A into t1
```

```
        li          $t2, 0x07f90932     # load the third quarter of vector A into t2
        li          $t3, 0x08a30889     # load the fourth quarter of vector A into t3
        li          $t4, 0x003D0035     # load the upper half of vector B into t4
        li          $t5, 0x0011019D     # load the lower half of vector B into t5

        srl         $t6, $t4, 16        # shift t6 and t7 left and right to set them to p
                                        # and q
        sll         $t7, $t4, 16
        srl         $t7, $t7, 16
        mult        $t6, $t7            # calculate and store n into t6
        mflo        $t6
        sll         $t7, $t5, 16        # store d into t7
        srl         $t7, $t7, 16

#       A's upper half
        srl         $t8, $t0, 16        # set t8 to the first element
        jal         Decry               # call the decryption subroutine
        add         $s0, $s0, $t8       # store the decrypted element into D's first
                                        # element
        sll         $s0, $s0, 8         # shift the element to make room for the next one

        sll         $t8, $t0, 16        # set t8 to the second element
        srl         $t8, $t8, 16
        jal         Decry               # call the decryption subroutine
        add         $s0, $s0, $t8       # store the decrypted element into D's second
                                        # element
        sll         $s0, $s0, 8         # shift the element to make room for the next one

        srl         $t8, $t1, 16        # set t8 to the third element
        jal         Decry               # call the decryption subroutine
        add         $s0, $s0, $t8       # store the decrypted element into D's third
                                        # element
        sll         $s0, $s0, 8         # shift the element to make room for the next one

        sll         $t8, $t1, 16        # set t8 to the fourth element
        srl         $t8, $t8, 16
        jal         Decry               # call the decryption subroutine
        add         $s0, $s0, $t8       # store the decrypted element into D's fourth
                                        # element

#       A's lower half
        srl         $t8, $t2, 16        # set t8 to the first element
        jal         Decry               # call the decryption subroutine
        add         $s1, $s1, $t8       # store the decrypted element into D's first
                                        # element
        sll         $s1, $s1, 8         # shift the element to make room for the next one

        sll         $t8, $t2, 16        # set t8 to the second element
```

```
        srl         $t8, $t8, 16
        jal         Decry               # call the decryption subroutine
        add         $s1, $s1, $t8       # store the decrypted element into D's second
                                        # element
        sll         $s1, $s1, 8         # shift the element to make room for the next one

        srl         $t8, $t3, 16        # set t8 to the third element
        jal         Decry               # call the decryption subroutine
        add         $s1, $s1, $t8       # store the decrypted element into D's third
                                        # element
        sll         $s1, $s1, 8         # shift the element to make room for the next one

        sll         $t8, $t3, 16        # set t8 to the fourth element
        srl         $t8, $t8, 16
        jal         Decry               # call the decryption subroutine
        add         $s1, $s1, $t8       # store the decrypted element into D's fourth
                                        # element

#       Function code for exit
exit:   ori         $v0, $zero, 10
        syscall

#       Subroutine implementing RSA decryption
Decry:  add         $a0, $zero, $zero   # resets a0 to zero
        addi        $a1, $zero, 0x1     # resets a1 to 1

Loop:   addi        $a0, $a0, 0x1       # increments the loop counter, a0
        mult        $a1, $t8            # multiplies a1 by the value being decrypted and
stores into a1
        mflo        $a1
        div         $a1, $t6            # divides a1 by n and stores into a1
        mfhi        $a1
        bne         $a0, $t7, Loop      # branches if a0 = t7, e
        add         $t8, $a1, $zero     # sets t8 to a1, the decrypted value
        jr          $ra                 # returns to main routine
```

2.5 Vec_encry

```
#*******************************************************************************
# File name:          vec_encry.asm
# Version:            1.0
# Date:               November 12, 2018
# Programmer:         Ellen Burger
#
# Description:        Code implementing the instruction 'vec_encry d, a, b' into MIPS
#                      architecture. A and B are 8-byte vectors; A has two-byte wide elements
#                      and B has four-byte wide elements. D is a 16-byte vector with four-byte
#                      wide elements. The upper and lower halves of A and B are split between
#                      two registers, and D quartered between four. The instruction encrypts
#                      each element from A using RSA encryption algorithm. The algorithm's p,
#                      q, e, and d values are supplied by each of B's elements. The encrypted
#                      elements of A are stored into D's elements one at a time.
#
# Register usage:     $t0 - contains the upper half of vector A
#                     $t1 - contains the lower half of vector A
#                     $t2 - contains the upper half of vector B, p in the upper element and q
#                     in the lower
#                     $t3 - contains the lower half of vector B, e in the upper element and d
#                     in the lower
#                     $t4 - contains n which is found by p * q
#                     $t5 - contains e
#                     $t6 - contains the element from A to be encrypted, then stores into D
#                     after encryption
#                     $s0 - contains the first quarter of vector D
#                     $s1 - contains the second quarter  of vector D
#                     $s2 - contains the third quarter of vector D
#                     $s3 - contains the fourth quarter of vector D
#                     $a0 - acts as a loop counter, ending the RSA algorithm when equaling e
#                     $a1 - used to calculate the encrypted value in the RSA algorithm
#                     $ra - holds the return address
#*******************************************************************************

# Main code segment
.text
.globl  main

main:  li            $t0, 0x5D23473C       # load the upper half of vector A into t0
       li            $t1, 0x087F196F       # load the lower half of vector A into t1
```

```
        li          $t2, 0x003D0035     # load the upper half of vector B into t2
        li          $t3, 0x0011019D     # load the lower half of vector B into t2

        srl         $t4, $t2, 16        # shift t4 and t5 left and right to set them to p
                                        # and q
        sll         $t5, $t2, 16
        srl         $t5, $t5, 16
        mult        $t4, $t5            # calculate and store n into t4
        mflo        $t4
        srl         $t5, $t3, 16        # store e into t5

#       A's upper half
        srl         $t6, $t0, 24        # set t6 to the first element
        jal         Encry               # call the encryption subroutine
        add         $s0, $s0, $t6       # store the encrypted element into D's first
                                        # element
        sll         $s0, $s0, 16        # shift the element to make room for the next one

        sll         $t6, $t0, 8         # set t6 to the second element
        srl         $t6, $t6, 24
        jal         Encry               # call the encryption subroutine
        add         $s0, $s0, $t6       # store the encrypted element into D's second
                                        # element

        sll         $t6, $t0, 16        # set t6 to the third element
        srl         $t6, $t6, 24
        jal         Encry               # call the encryption subroutine
        add         $s1, $s1, $t6       # store the encrypted element into D's third
                                        # element
        sll         $s1, $s1, 16        # shift the element to make room for the next one

        sll         $t6, $t0, 24        # set t6 to the fourth element
        srl         $t6, $t6, 24
        jal         Encry               # call the encryption subroutine
        add         $s1, $s1, $t6       # store the encrypted element into D's fourth
                                        # element

#       A's lower half
        srl         $t6, $t1, 24        # set t6 to the first element
        jal         Encry               # call the encryption subroutine
        add         $s2, $s2, $t6       # store the encrypted element into D's first
                                        # element
        sll         $s2, $s2, 16        # shift the element to make room for the next one

        sll         $t6, $t1, 8         # set t6 to the second element
        srl         $t6, $t6, 24
        jal         Encry               # call the encryption subroutine
        add         $s2, $s2, $t6       # store the encrypted element into D's second
```

```
                                        # element

        sll         $t6, $t1, 16        # set t6 to the third element
        srl         $t6, $t6, 24
        jal         Encry               # call the encryption subroutine
        add         $s3, $s3, $t6       # store the encrypted element into D's third
                                        # element
        sll         $s3, $s3, 16        # shift the element to make room for the next one

        sll         $t6, $t1, 24        # set t6 to the fourth element
        srl         $t6, $t6, 24
        jal         Encry               # call the encryption subroutine
        add         $s3, $s3, $t6       # store the encrypted element into D's fourth
                                        # element

#       Function code for exit
exit:   ori         $v0, $zero, 10
        syscall

#       Subroutine implementing RSA encryption
Encry:  add         $a0, $zero, $zero   # resets a0 to zero
        addi        $a1, $zero, 0x1     # resets a1 to 1

Loop:   addi        $a0, $a0, 0x1       # increments the loop counter, a0
        mult        $a1, $t6            # multiplies a1 by the value being encrypted and
                                        # stores into a1
        mflo        $a1
        div         $a1, $t4            # divides a1 by n and stores into a1
        mfhi        $a1
        bne         $a0, $t5, Loop      # branches if a0 = t5, e
        add         $t6, $a1, $zero     # sets t6 to a1, the encrypted value
        jr          $ra                 # returns to main routine
```

2.6 Vec_or

```
#*******************************************************************************
# File name:          vec_or.asm
# Version:            1.0
# Date:               November 12, 2018
# Programmer:         Ellen Burger
#
# Description:        Code implementing the instruction 'vec_or d, a, b' into MIPS
#                     architecture. D, A, and B are 8-byte vectors with 2-byte wide elements.
#                     The upper and lower halves of each vector are split between two
#                     registers. The instruction or's each element from A and B one at a time
#                     and stores it into the corresponding element in vector D.
#
# Register usage:     $t0 - contains the upper half of vector A
#                     $t1 - contains the lower half of vector A
#                     $t2 - contains the upper half of vector B
#                     $t3 - contains the lower half of vector B
#                     $t4 - contains vector A's two bytes that are being or'd, then receives
#                     the result to be stored into vector D. Set to 0xff if the result causes
#                     overflow
#                     $t5 - contains vector B's two bytes that are being or'd
#                     $s0 - contains the upper half of vector D
#                     $s1 - contains the lower half of vector D
#*******************************************************************************

# Main code segment
.text
.globl  main

main:   li          $t0, 0x233C475D       # load the upper half of vector A into t0
        li          $t1, 0x087F196F       # load the lower half of vector A into t1
        li          $t2, 0x981963C5       # load the upper half of vector B into t2
        li          $t3, 0x5E80B36E       # load the lower half of vector B into t3

#       Upper half of the vectors
        srl         $t4, $t0, 24          # shift t4 and t5 right to gain their first
                                          # elements
        srl         $t5, $t2, 24
        or          $t4, $t4, $t5         # or elements and store the result in t4
        add         $s0, $t4, $zero       # store result into s0, the destination vector D
        sll         $s0, $s0, 8           # shift the bytes twice to make room for the next
```

```
                                        # result

        sll         $t4, $t0, 8         # shift t4 and t5 left then right to gain their
                                        # second elements
        sll         $t5, $t2, 8
        srl         $t4, $t4, 24
        srl         $t5, $t5, 24
        or          $t4, $t4, $t5       # or elements and store the result in t4
        add         $s0, $s0, $t4       # store result into s0, the destination vector D
        sll         $s0, $s0, 8         # shift the bytes twice to make room for the next
                                        # result

        sll         $t4, $t0, 16        # shift t4 and t5 left then right to gain their
                                        # third elements
        sll         $t5, $t2, 16
        srl         $t4, $t4, 24
        srl         $t5, $t5, 24
        or          $t4, $t4, $t5       # or elements and store the result in t4
        add         $s0, $s0, $t4       # store result into s0, the destination vector D
        sll         $s0, $s0, 8         # shift the bytes twice to make room for the next
                                        # result

        sll         $t4, $t0, 24        # shift t4 and t5 left then right to gain their
                                        # fourth elements
        sll         $t5, $t2, 24
        srl         $t4, $t4, 24
        srl         $t5, $t5, 24
        or          $t4, $t4, $t5       # or elements and store the result in t4
        add         $s0, $s0, $t4       # store result into s0, the destination vector D

#       Lower half of the vectors
        srl         $t4, $t1, 24        # shift t4 and t5 right to gain their fifth
                                        # elements
        srl         $t5, $t3, 24
        or          $t4, $t4, $t5       # or elements and store the result in t4
        add         $s1, $t4, $zero     # store result into s1, the destination vector D
        sll         $s1, $s1, 8         # shift the bytes twice to make room for the next
                                        # result

        sll         $t4, $t1, 8         # shift t4 and t5 left then right to gain their
                                        # sixth elements
        sll         $t5, $t3, 8
        srl         $t4, $t4, 24
        srl         $t5, $t5, 24
        or          $t4, $t4, $t5       # or elements and store the result in t4
        add         $s1, $s1, $t4       # store result into s1, the destination vector D
        sll         $s1, $s1, 8         # shift the bytes twice to make room for the next
```

```
                                        # result

        sll         $t4, $t1, 16        # shift t4 and t5 left then right to gain their
                                        # seventh elements
        sll         $t5, $t3, 16
        srl         $t4, $t4, 24
        srl         $t5, $t5, 24
        or          $t4, $t4, $t5       # or elements and store the result in t4
        add         $s1, $s1, $t4       # store result into s1, the destination vector D
        sll         $s1, $s1, 8         # shift the bytes twice to make room for the next
                                        # result

        sll         $t4, $t1, 24        # shift t4 and t5 left then right to gain their
                                        # eighth elements
        sll         $t5, $t3, 24
        srl         $t4, $t4, 24
        srl         $t5, $t5, 24
        or          $t4, $t4, $t5       # or elements and store the result in t4
        add         $s1, $s1, $t4       # store result into s1, the destination vector D

#       Function code for exit
exit:   ori         $v0, $zero, 10
        syscall
```

2.7 Vec_sortlow

```
#*******************************************************************************
# File name:          vec_sortlow.asm
# Version:            1.0
# Date:               November 12, 2018
# Programmer:         Ellen Burger
#
# Description:        Code implementing the instruction 'vec_sortlow d, a' into MIPS
#                     architecture. D and A are 8-byte vectors with 2-byte wide elements. The
#                     upper and lower halves of each vector are split between two registers.
#                     The instruction sorts the elements in A until the entire vector is in
#                     decreasing numerical order. The sorted result is stored in D.
#
# Register usage:     $t0 - contains the upper half of vector A
#                     $t1 - contains the lower half of vector A
#                     $t2 - contains an element of A to be compared with the following
#                     element, then swapped if less than
#                     $t3 - contains an element of A to be compared with the previous
#                     element, then swapped if not less than
#                     $t4 - contains a portion of s0 or s1 to be subtracted from s0 or s1.
#                     The gap is then filled in with t1 and t2, swapped if needing to be
#                     sorted
#                     $s0 - contains the upper half of vector D
#                     $s1 - contains the lower half of vector D
#                     $a0 - set to 0 if t2 < t3 and needs to be swapped
#                     $ra - holds the return address
#*******************************************************************************

# Main code segment
.text
.globl  main

main:   li          $t0, 0x5D23473C       # load the upper half of vector A into t0
        li          $t1, 0x087F196F       # load the lower half of vector A into t1

        add         $s0, $t0, $zero       # set s0 to t0, the vector to be sorted
        add         $s1, $t1, $zero       # set s1 to t1, the vector to be sorted

L1:     add         $t9, $zero, $zero     # sets the swap checker back to zero
        srl         $t2, $s0, 24          # shift t2 and t3 left then right to gain the
```

```
                                        # first and second elements
        sll         $t3, $s0, 8
        srl         $t3, $t3, 24
        slt         $a0, $t3, $t2       # checks if the element in t2 is greater than
                                        # t3's, branches to L2 if so
        bne         $a0, $zero, L2
        sll         $s0, $s0, 16        # if not, swaps the elements
        srl         $s0, $s0, 16
        jal         Swap
        sll         $t2, $t2, 24
        sll         $t3, $t3, 16
        add         $s0, $s0, $t2
        add         $s0, $s0, $t3

L2:     sll         $t2, $s0, 8         # shift t2 and t3 left then right to gain the
                                        # second and third elements
        sll         $t3, $s0, 16
        srl         $t2, $t2, 24
        srl         $t3, $t3, 24
        slt         $a0, $t3, $t2       # checks if the element in t2 is greater than
                                        # t3's, branches to L3 if so
        bne         $a0, $zero, L3
        sll         $t4, $s0, 8         # if not, swaps the elements
        srl         $t4, $t4, 16
        sll         $t4, $t4, 8
        sub         $s0, $s0, $t4
        jal         Swap
        sll         $t2, $t2, 16
        sll         $t3, $t3, 8
        add         $s0, $s0, $t2
        add         $s0, $s0, $t3

L3:     sll         $t2, $s0, 16        # shift t2 and t3 left then right to gain the
                                        # third and fourth elements
        sll         $t3, $s0, 24
        srl         $t2, $t2, 24
        srl         $t3, $t3, 24
        slt         $a0, $t3, $t2       # checks if the element in t2 is greater than
                                        # t3's, branches to L4 if so
        bne         $a0, $zero, L4
        srl         $s0, $s0, 16        # if not, swaps the elements
        sll         $s0, $s0, 16
        jal         Swap
        sll         $t2, $t2, 8
        add         $s0, $s0, $t2
        add         $s0, $s0, $t3

L4:     sll         $t2, $s0, 24        # shift t2 and t3 left then right to gain the
```

```
                                        # fourth and fifth elements
        srl         $t2, $t2, 24
        srl         $t3, $s1, 24
        slt         $a0, $t3, $t2       # checks if the element in t2 is greater than
                                        # t3's, branches to L5 if so
        bne         $a0, $zero, L5
        srl         $s0, $s0, 8         # if not, swaps the elements
        sll         $s0, $s0, 8
        sll         $s1, $s1, 8
        srl         $s1, $s1, 8
        jal         Swap
        sll         $t3, $t3, 24
        add         $s0, $s0, $t2
        add         $s1, $s1, $t3

L5:     srl         $t2, $s1, 24        # shift t2 and t3 left then right to gain the
                                        # fifth and sixth elements
        sll         $t3, $s1, 8
        srl         $t3, $t3, 24
        slt         $a0, $t3, $t2       # checks if the element in t2 is greater than
                                        # t3's, branches to L6 if so
        bne         $a0, $zero, L6
        sll         $s1, $s1, 16        # if not, swaps the elements
        srl         $s1, $s1, 16
        jal         Swap
        sll         $t2, $t2, 24
        sll         $t3, $t3, 16
        add         $s1, $s1, $t2
        add         $s1, $s1, $t3

L6:     sll         $t2, $s1, 8         # shift t2 and t3 left then right to gain the
                                        # sixth and seventh elements
        sll         $t3, $s1, 16
        srl         $t2, $t2, 24
        srl         $t3, $t3, 24
        slt         $a0, $t3, $t2       # checks if the element in t2 is greater than
                                        # t3's, branches to L7 if so
        bne         $a0, $zero, L7
        sll         $t4, $s1, 8         # if not, swaps the elements
        srl         $t4, $t4, 16
        sll         $t4, $t4, 8
        sub         $s1, $s1, $t4
        jal         Swap
        sll         $t2, $t2, 16
        sll         $t3, $t3, 8
        add         $s1, $s1, $t2
        add         $s1, $s1, $t3
```

```
L7:     sll         $t2, $s1, 16        # shift t2 and t3 left then right to gain the
                                        # seventh and eigth elements
        sll         $t3, $s1, 24
        srl         $t2, $t2, 24
        srl         $t3, $t3, 24
        slt         $a0, $t3, $t2       # checks if the element in t2 is greater than
                                        # t3's, branches to L4 if so
        bne         $a0, $zero, L8
        srl         $s1, $s1, 16        # if not, swaps the elements
        sll         $s1, $s1, 16
        jal         Swap
        sll         $t2, $t2, 8
        add         $s1, $s1, $t2
        add         $s1, $s1, $t3

L8:     bne         $t9, $zero, L1      # if no swaps have happened the entire sequence,
                                        # the vector is sorted and instruction ended

#       Function code for exit
exit:   ori         $v0, $zero, 10
        syscall

#       Subroutine implementing the swap
Swap:   add         $a0, $t2, $zero     # swaps the registers the elements are in
        add         $t2, $t3, $zero
        add         $t3, $a0, $zero
        addi        $t9, $zero, 0x1     # t9 is set to 1 if a swap has taken place
        jr          $ra                 # returns to main routine
```

2.8 Vec_sortup

```
#*******************************************************************************
# File name:          vec_sortup.asm
# Version:            1.0
# Date:               November 12, 2018
# Programmer:         Ellen Burger
#
# Description:        Code implementing the instruction 'vec_sortup d, a' into MIPS
#                     architecture. D and A are 8-byte vectors with 2-byte wide elements. The
#                     upper and lower halves of each vector are split between two registers.
#                     The instruction sorts the elements in A until the entire vector is in
#                     increasing numerical order. The sorted result is stored in D.
#
# Register usage:     $t0 - contains the upper half of vector A
#                     $t1 - contains the lower half of vector A
#                     $t2 - contains an element of A to be compared with the following
#                     element, then swapped if not less than
#                     $t3 - contains an element of A to be compared with the previous
#                     element, then swapped if less than
#                     $t4 - contains a portion of s0 or s1 to be subtracted from s0 or s1.
#                     The gap is then filled in with t1 and t2, swapped if needing to be
#                     sorted
#                     $s0 - contains the upper half of vector D
#                     $s1 - contains the lower half of vector D
#                     $a0 - set to 0 if t2 > t3 and needs to be swapped
#                     $ra - holds the return address
#*******************************************************************************

# Main code segment
.text
.globl  main

main:   li          $t0, 0x5D23473C       # load the upper half of vector A into t0
        li          $t1, 0x087F196F       # load the lower half of vector A into t1

        add         $s0, $t0, $zero       # set s0 to t0, the vector to be sorted
        add         $s1, $t1, $zero       # set s1 to t1, the vector to be sorted

L1:     add         $t9, $zero, $zero     # sets the swap checker back to zero
        srl         $t2, $s0, 24          # shift t2 and t3 left then right to gain the
first and second elements
```

```
        sll         $t3, $s0, 8
        srl         $t3, $t3, 24
        slt         $a0, $t2, $t3       # checks if the element in t2 is less than t3's,
                                        # branches to L2 if so
        bne         $a0, $zero, L2
        sll         $s0, $s0, 16        # if not, swaps the elements
        srl         $s0, $s0, 16
        jal         Swap
        sll         $t2, $t2, 24
        sll         $t3, $t3, 16
        add         $s0, $s0, $t2
        add         $s0, $s0, $t3

L2:     sll         $t2, $s0, 8         # shift t2 and t3 left then right to gain the
                                        # second and third elements
        sll         $t3, $s0, 16
        srl         $t2, $t2, 24
        srl         $t3, $t3, 24
        slt         $a0, $t2, $t3       # checks if the element in t2 is less than t3's,
                                        # branches to L3 if so
        bne         $a0, $zero, L3
        sll         $t4, $s0, 8         # if not, swaps the elements
        srl         $t4, $t4, 16
        sll         $t4, $t4, 8
        sub         $s0, $s0, $t4
        jal         Swap
        sll         $t2, $t2, 16
        sll         $t3, $t3, 8
        add         $s0, $s0, $t2
        add         $s0, $s0, $t3

L3:     sll         $t2, $s0, 16        # shift t2 and t3 left then right to gain the
                                        # third and fourth elements
        sll         $t3, $s0, 24
        srl         $t2, $t2, 24
        srl         $t3, $t3, 24
        slt         $a0, $t2, $t3       # checks if the element in t2 is less than t3's,
                                        # branches to L4 if so
        bne         $a0, $zero, L4
        srl         $s0, $s0, 16        # if not, swaps the elements
        sll         $s0, $s0, 16
        jal         Swap
        sll         $t2, $t2, 8
        add         $s0, $s0, $t2
        add         $s0, $s0, $t3

L4:     sll         $t2, $s0, 24        # shift t2 and t3 left then right to gain the
```

```
                                        # fourth and fifth elements
        srl         $t2, $t2, 24
        srl         $t3, $s1, 24
        slt         $a0, $t2, $t3       # checks if the element in t2 is less than t3's,
                                        # branches to L5 if so
        bne         $a0, $zero, L5
        srl         $s0, $s0, 8         # if not, swaps the elements
        sll         $s0, $s0, 8
        sll         $s1, $s1, 8
        srl         $s1, $s1, 8
        jal         Swap
        sll         $t3, $t3, 24
        add         $s0, $s0, $t2
        add         $s1, $s1, $t3

L5:     srl         $t2, $s1, 24        # shift t2 and t3 left then right to gain the
                                        # fifth and sixth elements
        sll         $t3, $s1, 8
        srl         $t3, $t3, 24
        slt         $a0, $t2, $t3       # checks if the element in t2 is less than t3's,
                                        #branches to L6 if so
        bne         $a0, $zero, L6
        sll         $s1, $s1, 16        # if not, swaps the elements
        srl         $s1, $s1, 16
        jal         Swap
        sll         $t2, $t2, 24
        sll         $t3, $t3, 16
        add         $s1, $s1, $t2
        add         $s1, $s1, $t3

L6:     sll         $t2, $s1, 8         # shift t2 and t3 left then right to gain the
                                        # sixth and seventh elements
        sll         $t3, $s1, 16
        srl         $t2, $t2, 24
        srl         $t3, $t3, 24
        slt         $a0, $t2, $t3       # checks if the element in t2 is less than t3's,
                                        # branches to L7 if so
        bne         $a0, $zero, L7
        sll         $t4, $s1, 8         # if not, swaps the elements
        srl         $t4, $t4, 16
        sll         $t4, $t4, 8
        sub         $s1, $s1, $t4
        jal         Swap
        sll         $t2, $t2, 16
        sll         $t3, $t3, 8
        add         $s1, $s1, $t2
        add         $s1, $s1, $t3
```

```
L7:     sll         $t2, $s1, 16        # shift t2 and t3 left then right to gain the
                                        # seventh and eigth elements
        sll         $t3, $s1, 24
        srl         $t2, $t2, 24
        srl         $t3, $t3, 24
        slt         $a0, $t2, $t3       # checks if the element in t2 is less than t3's,
                                        # branches to L4 if so
        bne         $a0, $zero, L8
        srl         $s1, $s1, 16        # if not, swaps the elements
        sll         $s1, $s1, 16
        jal         Swap
        sll         $t2, $t2, 8
        add         $s1, $s1, $t2
        add         $s1, $s1, $t3

L8:     bne         $t9, $zero, L1      # if no swaps have happened the entire sequence,
                                        # the vector is sorted and instruction ended

#       Function code for exit
exit:   ori         $v0, $zero, 10
        syscall

#       Subroutine implementing the swap
Swap:   add         $a0, $t2, $zero     # swaps the registers the elements are in
        add         $t2, $t3, $zero
        add         $t3, $a0, $zero
        addi        $t9, $zero, 0x1     # t9 is set to 1 if a swap has taken place
        jr          $ra                 # returns to main routine
```

2.9 Vec_subsu

```
#*******************************************************************************
# File name:          vec_subsu.asm
# Version:            1.0
# Date:               November 12, 2018
# Programmer:         Ellen Burger
#
# Description:        Code implementing the instruction 'vec_subsu d, a, b' into MIPS
#                      architecture. D, A, and B are 8-byte vectors with 2-byte wide elements.
#                      The upper and lower halves of each vector are split between two
#                      registers. The instruction subtracts each element from A and B one at a
#                      time and stores it into the corresponding element in vector D.
#
# Register usage:     $t0 - contains the upper half of vector A
#                     $t1 - contains the lower half of vector A
#                     $t2 - contains the upper half of vector B
#                     $t3 - contains the lower half of vector B
#                     $t4 - contains vector A's two bytes that are being subtracted, then
#                      receives the difference to be stored into vector D. Set to 0x00 if the
#                      difference causes overflow
#                     $t5 - contains vector B's two bytes that are being subtracted
#                     $s0 - contains the upper half of vector D
#                     $s1 - contains the lower half of vector D
#                     $a0 - set to 0 if there is a difference overflow
#                     $ra - holds the return subress
#*******************************************************************************

# Main code segment
.text
.globl  main

main:  li            $t0, 0x233C475D       # load the upper half of vector A into t0
       li            $t1, 0x087F196F       # load the lower half of vector A into t1
       li            $t2, 0x981963C5       # load the upper half of vector B into t2
       li            $t3, 0x5E80B36E       # load the lower half of vector B into t3

#      Upper half of the vectors
       srl           $t4, $t0, 24          # shift t4 and t5 right to gain their first
                                           # elements
       srl           $t5, $t2, 24
       sub           $t4, $t4, $t5         # sub elements and store the result in t4
```

```
        jal         Check               # jump to subroutine to check for overflow
        add         $s0, $t4, $zero     # store difference into s0, the destination
                                        # vector D
        sll         $s0, $s0, 8         # shift the bytes twice to make room for the next
                                        # difference
        sll         $t4, $t0, 8         # shift t4 and t5 left then right to gain their
                                        # second elements
        sll         $t5, $t2, 8
        srl         $t4, $t4, 24
        srl         $t5, $t5, 24
        sub         $t4, $t4, $t5       # sub elements and store the result in t4
        jal         Check               # jump to subroutine to check for overflow
        add         $s0, $s0, $t4       # store difference into s0, the destination vector
                                        # D
        sll         $s0, $s0, 8         # shift the bytes twice to make room for the next
                                        # difference

        sll         $t4, $t0, 16        # shift t4 and t5 left then right to gain their
                                        # third elements
        sll         $t5, $t2, 16
        srl         $t4, $t4, 24
        srl         $t5, $t5, 24
        sub         $t4, $t4, $t5       # sub elements and store the result in t4
        jal         Check               # jump to subroutine to check for overflow
        add         $s0, $s0, $t4       # store difference into s0, the destination vector
                                        # D
        sll         $s0, $s0, 8         # shift the bytes twice to make room for the next
                                        # difference

        sll         $t4, $t0, 24        # shift t4 and t5 left then right to gain their
                                        # fourth elements
        sll         $t5, $t2, 24
        srl         $t4, $t4, 24
        srl         $t5, $t5, 24
        sub         $t4, $t4, $t5       # sub elements and store the result in t4
        jal         Check               # jump to subroutine to check for overflow
        add         $s0, $s0, $t4       # store difference into s0, the destination vector
                                        # D

#       Lower half of the vectors
        srl         $t4, $t1, 24        # shift t4 and t5 right to gain their fifth
                                        # elements
        srl         $t5, $t3, 24
        sub         $t4, $t4, $t5       # sub elements and store the result in t4
        jal         Check               # jump to subroutine to check for overflow
        add         $s1, $t4, $zero     # store difference into s1, the destination vector
                                        # D
        sll         $s1, $s1, 8         # shift the bytes twice to make room for the next
```

```
                                        # difference

        sll         $t4, $t1, 8         # shift t4 and t5 left then right to gain their
                                        # sixth elements
        sll         $t5, $t3, 8
        srl         $t4, $t4, 24
        srl         $t5, $t5, 24
        sub         $t4, $t4, $t5       # sub elements and store the result in t4
        jal         Check               # jump to subroutine to check for overflow
        add         $s1, $s1, $t4       # store difference into s1, the destination vector
                                        # D
        sll         $s1, $s1, 8         # shift the bytes twice to make room for the next
                                        # difference

        sll         $t4, $t1, 16        # shift t4 and t5 left then right to gain their
                                        # seventh elements
        sll         $t5, $t3, 16
        srl         $t4, $t4, 24
        srl         $t5, $t5, 24
        sub         $t4, $t4, $t5       # sub elements and store the result in t4
        jal         Check               # jump to subroutine to check for overflow
        add         $s1, $s1, $t4       # store difference into s1, the destination vector
                                        # D
        sll         $s1, $s1, 8         # shift the bytes twice to make room for the next
                                        # difference

        sll         $t4, $t1, 24        # shift t4 and t5 left then right to gain their
                                        # eighth elements
        sll         $t5, $t3, 24
        srl         $t4, $t4, 24
        srl         $t5, $t5, 24
        sub         $t4, $t4, $t5       # sub elements and store the result in t4
        jal         Check               # jump to subroutine to check for overflow
        add         $s1, $s1, $t4       # store difference into s1, the destination vector
                                        # D

#       Function code for exit
exit:   ori         $v0, $zero, 10
        syscall

#       Subroutine to check for overflow
Check:  slti        $a0, $t4, 0x0       # check if t4 is less than zero causing overflow
        beq         $a0, $zero, End     # exits subroutine if not
        add         $t4, $zero, $zero   # sets $t4 to zero if there's overflow
End:    jr          $ra                 # return to main routine
```

B. Output

1. Baseline SIMD Enhancements

The following snapshots showcase the “log file” with the registers before and after implementation of each **Baseline SIMD** enhancements, verifying the instructions are being fetched and executed correctly.

COLOR CODE

VECTOR	COLOR
a	
b	
c	
d	

1.1 Vec_addsu d, a, b

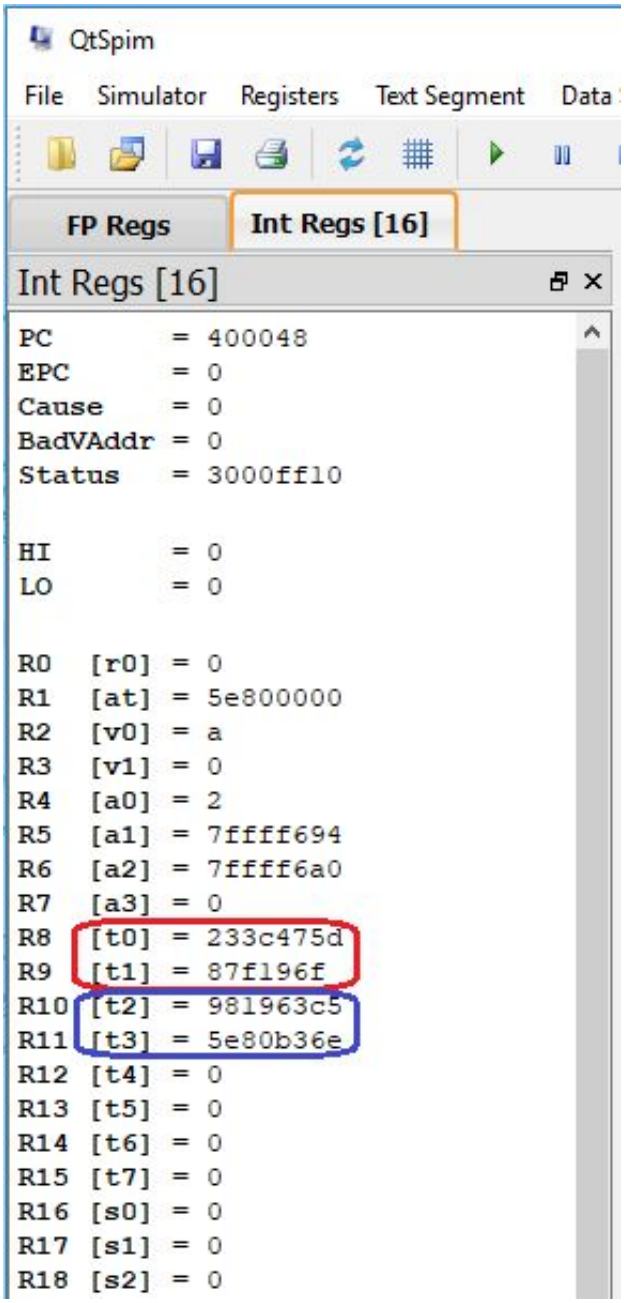


Figure: Before

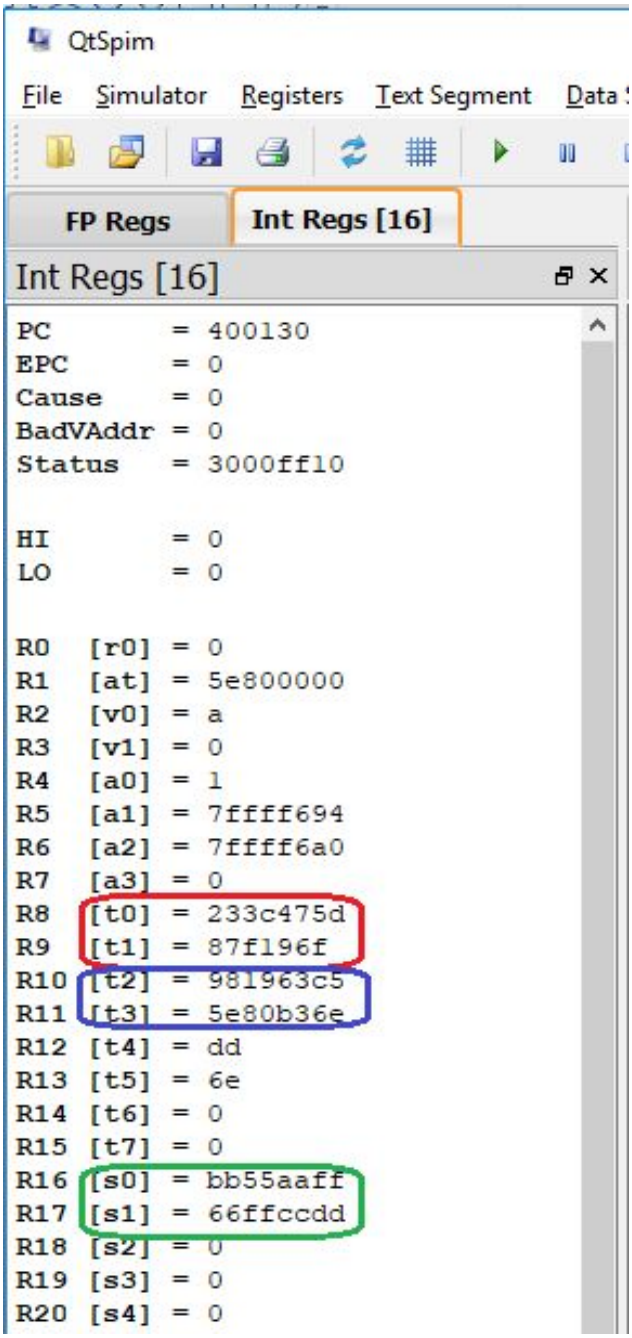


Figure: After

1.2 Vec_madd d, a, b, c

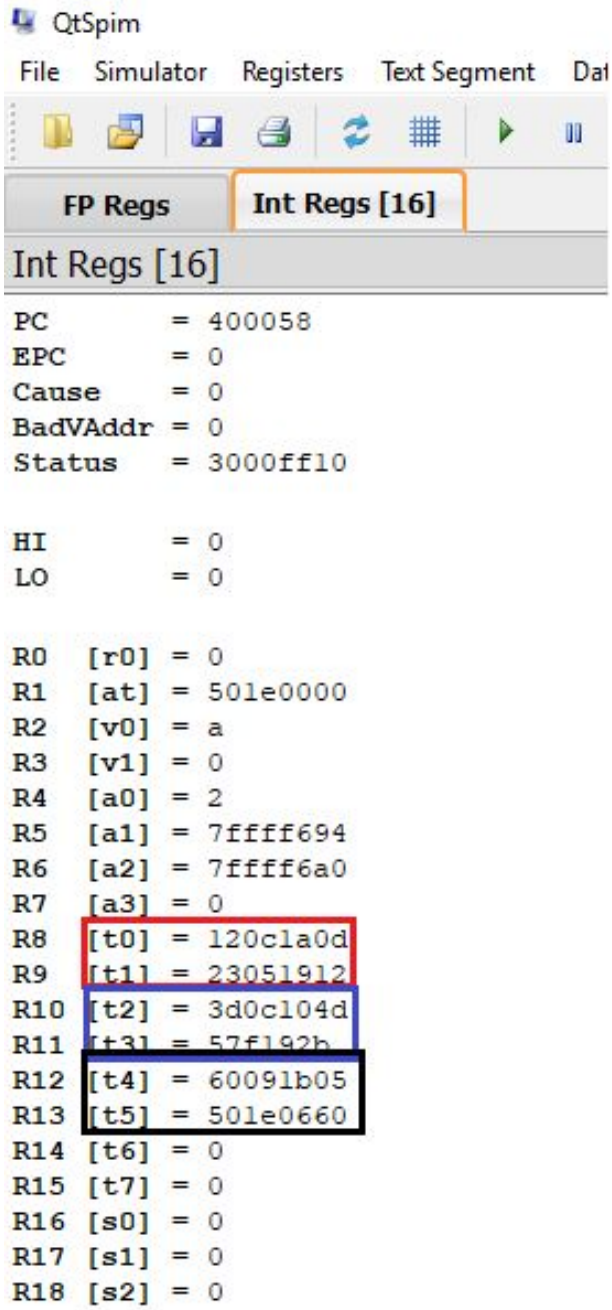


Figure: Before

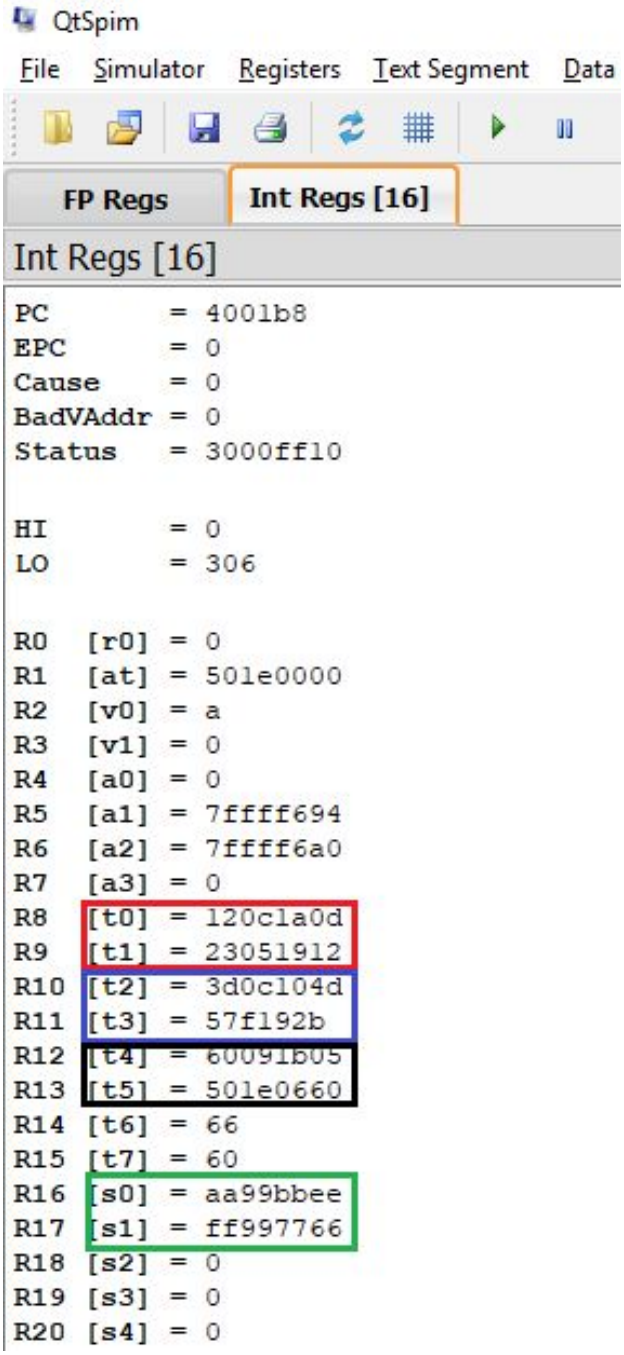


Figure: After

1.3 Vec_mule d, a, b

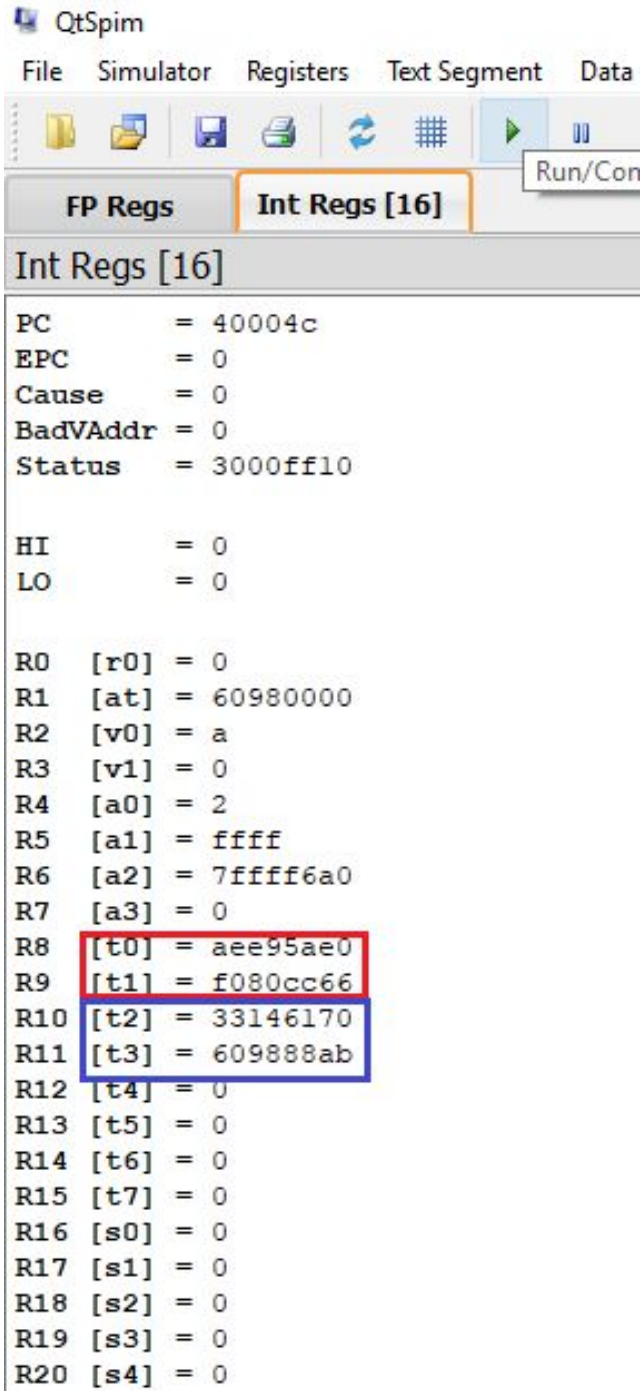


Figure: Before

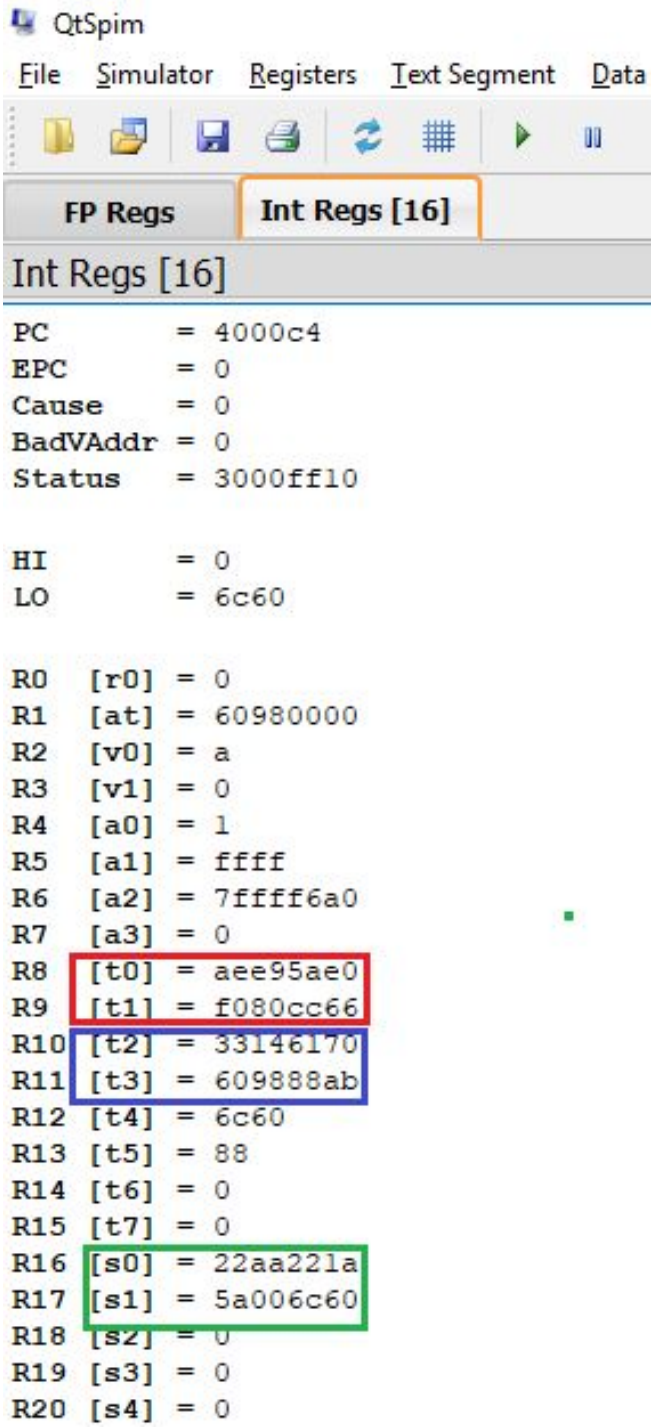


Figure: After

1.4 Vec_mulo d, a, b

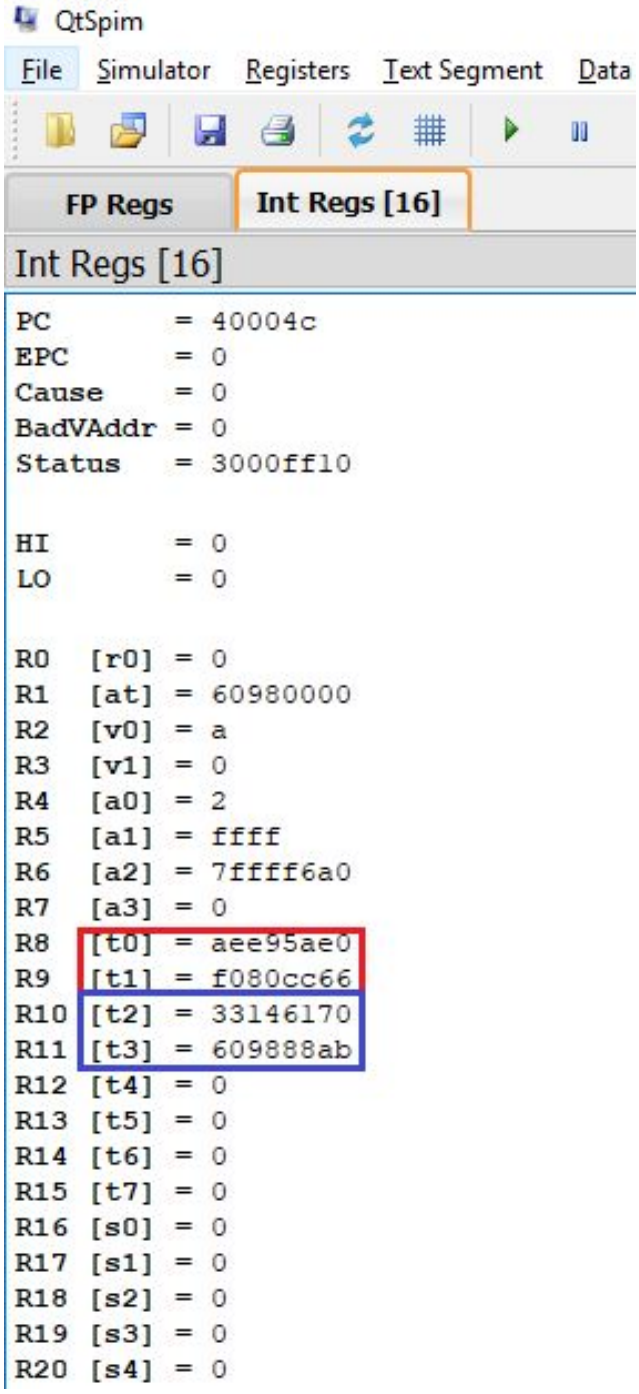


Figure: Before

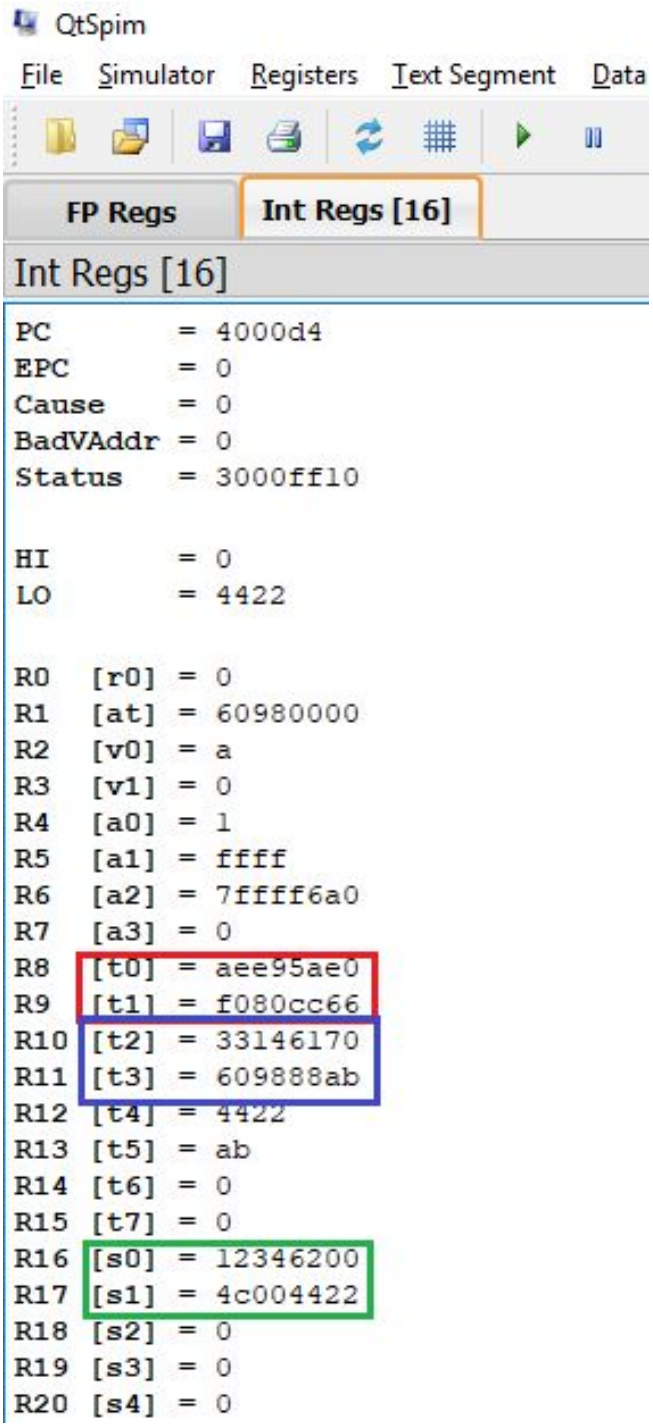


Figure: After

1.5 Vec_msums d, a, b,c

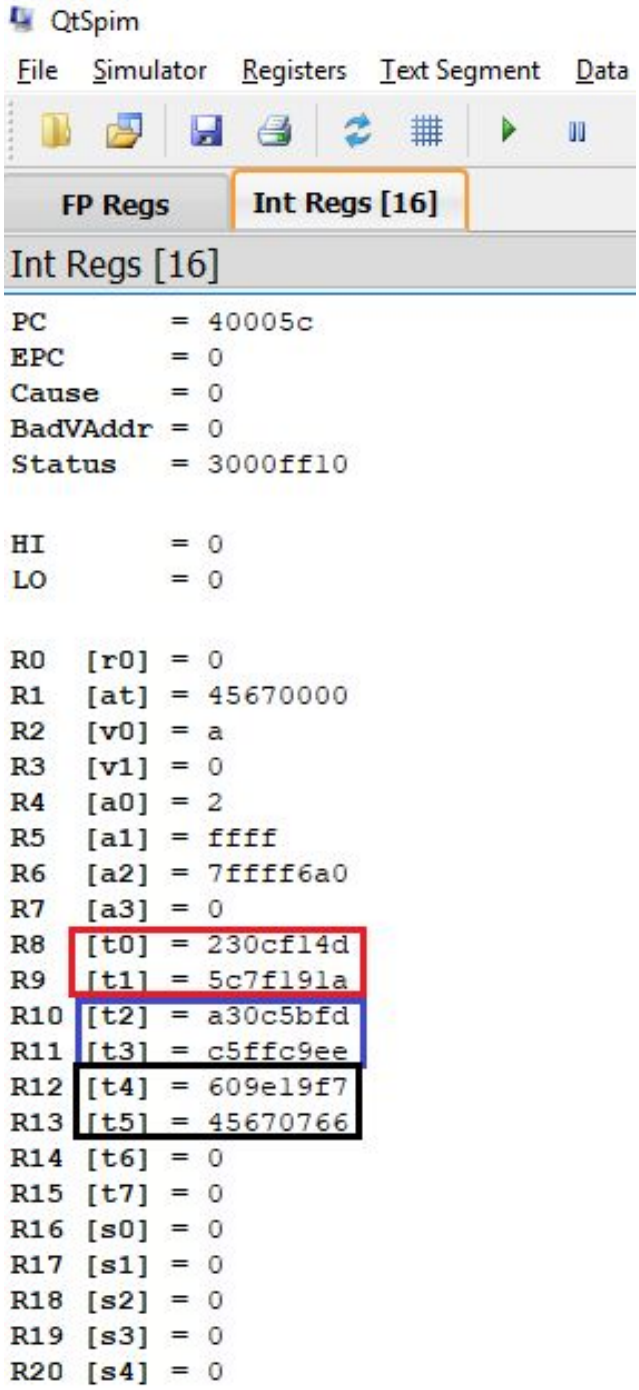


Figure: Before

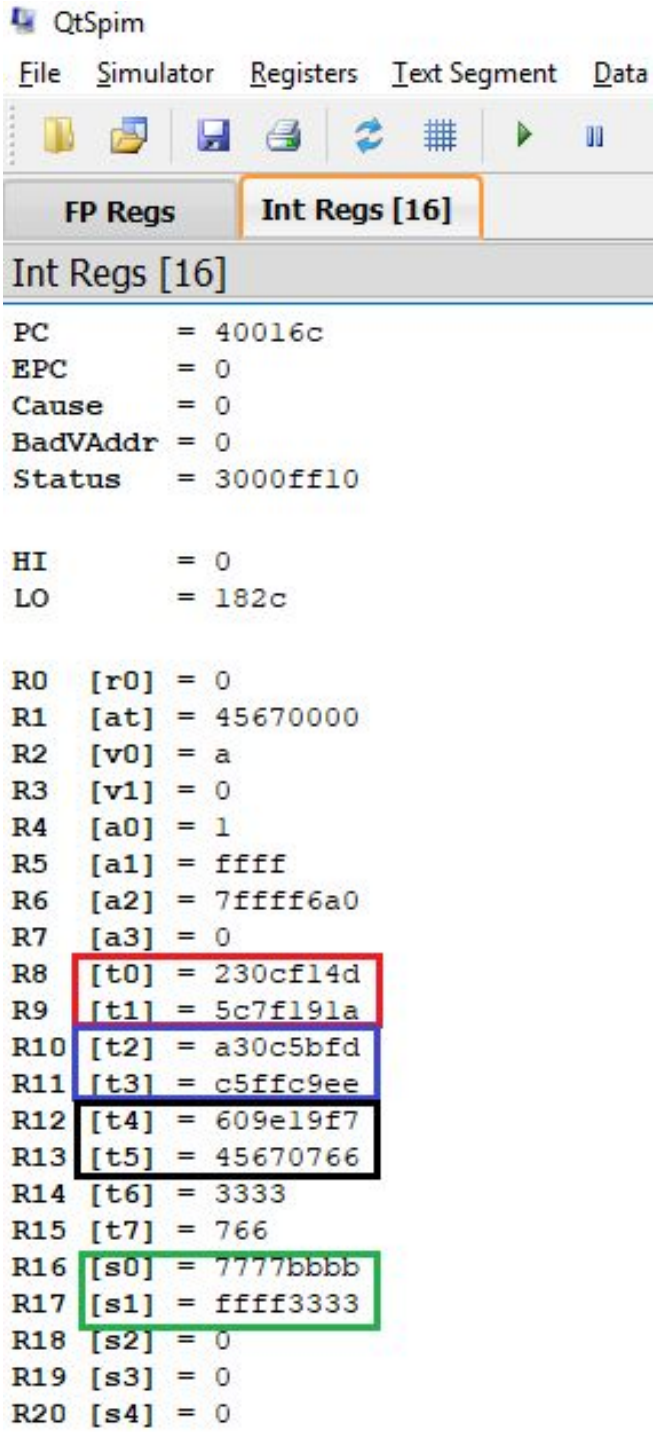


Figure: After

1.6 Vec_splat d, a, b

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x057f0000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x230c124d
$t1	9	0x057f192a
$t2	10	0x00000000
$t3	11	0x00000005
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000

Figure: Before

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000004
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000004
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x230c124d
$t1	9	0x057f192a
$t2	10	0x00000000
$t3	11	0x00000005
$t4	12	0x0000007f
$t5	13	0x00000004
$t6	14	0x00000006
$t7	15	0x00000000
$s0	16	0x7f7f7f7f
$s1	17	0x7f7f7f7f
$s2	18	0x00000000
$s3	19	0x00000000

Figure: After

1.7 Vec_mergel d,a,b

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0xcd230000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x5af0a501
$t1	9	0xab0155c3
$t2	10	0xa50f5a23
$t3	11	0xcd23aa3c
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000

Figure: Before

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0xcd230000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x5af0a501
$t1	9	0xab0155c3
$t2	10	0xa50f5a23
$t3	11	0xcd23aa3c
$t4	12	0x000000c3
$t5	13	0x0000003c
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0xabcd0123
$s1	17	0x55aac33c
$s2	18	0x00000000
$s3	19	0x00000000

Figure: After

1.8 Vec_mergeh d,a,b

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0xcd230000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x5af0a501
$t1	9	0xab0155c3
$t2	10	0xa50f5a23
$t3	11	0xcd23aa3c
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000

Figure: Before

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0xcd230000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x5af0a501
$t1	9	0xab0155c3
$t2	10	0xa50f5a23
$t3	11	0xcd23aa3c
$t4	12	0x00000001
$t5	13	0x00000023
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x5aa5f00f
$s1	17	0xa55a0123
$s2	18	0x00000000
$s3	19	0x00000000

Figure: After

1.9 Vec_pack d,a,b

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0xa6570000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x5afb6c1d
$t1	9	0xae5fc041
$t2	10	0x52f3a415
$t3	11	0xa657c849
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000

Figure: Before

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0xa6570000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x5afb6c1d
$t1	9	0xae5fc041
$t2	10	0x52f3a415
$t3	11	0xa657c849
$t4	12	0x00000089
$t5	13	0x00000009
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0xabcdef01
$s1	17	0x23456789
$s2	18	0x00000000
$s3	19	0x00000000

Figure: After

1.10 Vec_perm d,a,b,c

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x13050000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0xa567013d
$t1	9	0xab45393c
$t2	10	0xefc54d23
$t3	11	0x1277aacd
$t4	12	0x04171002
$t5	13	0x13050105
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000

Figure: Before

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000004
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000045
$a1	5	0x00000003
$a2	6	0x00000000
$a3	7	0x00000001
$t0	8	0xa567013d
$t1	9	0xab45393c
$t2	10	0xefc54d23
$t3	11	0x1277aacd
$t4	12	0x04171002
$t5	13	0x13050105
$t6	14	0x00000005
$t7	15	0x00000000
$s0	16	0xabcdef01
$s1	17	0x23456745
$s2	18	0x00000000
$s3	19	0x00000000

Figure: After

1.11 Vec_cmpeq d,a,b

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0xae5f0000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x5afb6c1d
$t1	9	0xa65fc040
$t2	10	0x52fba415
$t3	11	0xae5fc841
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000

Figure: Before

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0xae5f0000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x5afb6c1d
$t1	9	0xa65fc040
$t2	10	0x52fba415
$t3	11	0xae5fc841
$t4	12	0x00000040
$t5	13	0x00000041
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00ff0000
$s1	17	0x00ff0000
$s2	18	0x00000000
$s3	19	0x00000000

Figure: After

1.12 Vec_cmpltu d,a,b

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0xae5f0000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x5afb6c1d
$t1	9	0xa65fc040
$t2	10	0x52fba415
$t3	11	0xae5fc841
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000

Figure: Before

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0xae5f0000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x5afb6c1d
$t1	9	0xa65fc040
$t2	10	0x52fba415
$t3	11	0xae5fc841
$t4	12	0x00000001
$t5	13	0x00000041
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x0000ff00
$s1	17	0xff00ffff
$s2	18	0x00000000
$s3	19	0x00000000

Figure: After

2. Specific Application Enhancements

2.1 vec_and d,a,b

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x5e800000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x233c475d
$t1	9	0x087f196f
$t2	10	0x981963c5
$t3	11	0x5e80b36e
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000

Figure: Before

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x5e800000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x233c475d
$t1	9	0x087f196f
$t2	10	0x981963c5
$t3	11	0x5e80b36e
$t4	12	0x0000006e
$t5	13	0x0000006e
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x001843dd
$s1	17	0x0800116e
$s2	18	0x00000000
$s3	19	0x00000000

Figure: After

2.2 vec_beq a,b,imm

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x087f0000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x233c475d
$t1	9	0x087f196f
$t2	10	0x233c475d
$t3	11	0x087f196f
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400034
hi		0x00000000
lo		0x00000000

Figure:

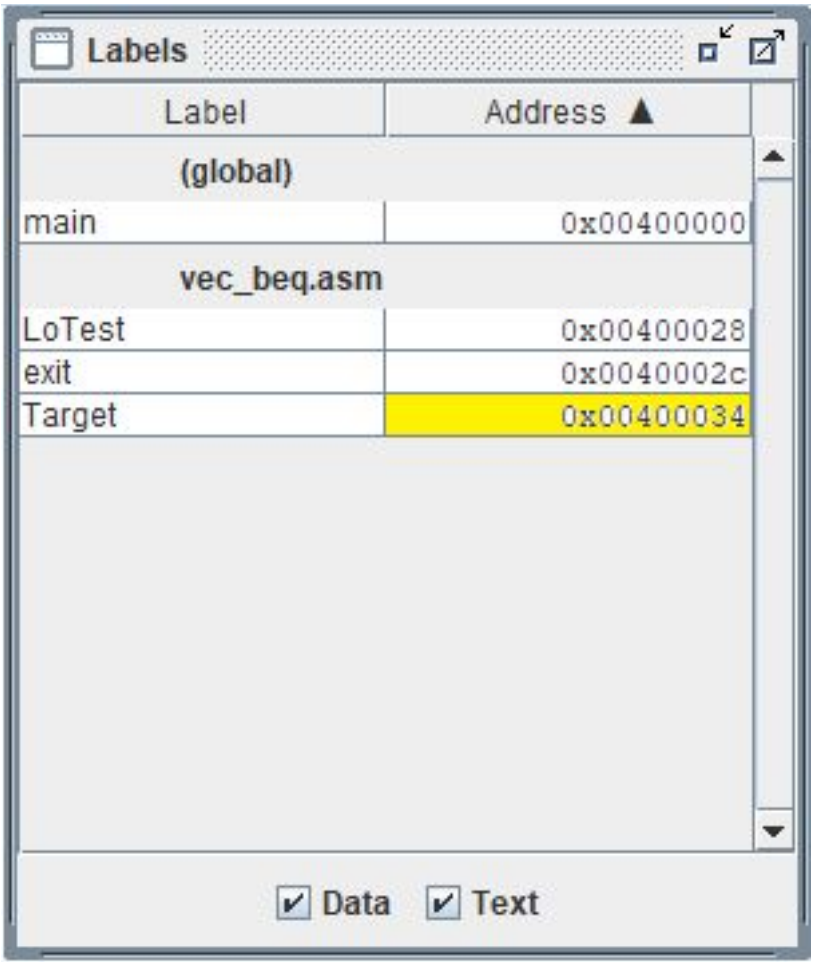

Labels
Label
Address
(global)
main
0x00400000
vec_beq.asm
LoTest
0x00400028
exit
0x0040002c
Target
0x00400034
Data
Text

Figure:

2.3 vec_bne a,b,imm

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x087f0000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x233c475d
$t1	9	0x087f196f
$t2	10	0x233c475d
$t3	11	0x087f087f
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400030
hi		0x00000000
lo		0x00000000

Figure:

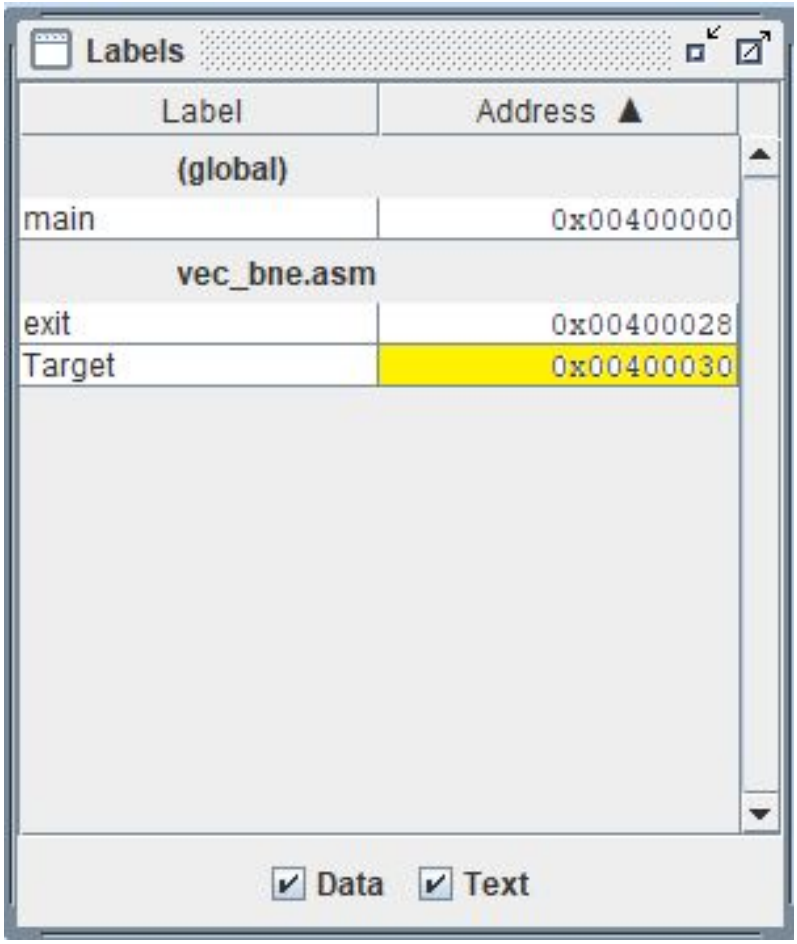

Labels
Label
Address
(global)
main
0x00400000
vec_bne.asm
exit
0x00400028
Target
0x00400030
Data
Text

Figure:

2.4 vec_decry d, a, b

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00110000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x024d0399
$t1	9	0x029d0449
$t2	10	0x07f90932
$t3	11	0x08a30889
$t4	12	0x003d0035
$t5	13	0x0011019d
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000

Figure: Before

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00110000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x0000019d
$a1	5	0x0000006f
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x024d0399
$t1	9	0x029d0449
$t2	10	0x07f90932
$t3	11	0x08a30889
$t4	12	0x003d0035
$t5	13	0x0011019d
$t6	14	0x00000ca1
$t7	15	0x0000019d
$s0	16	0x5d23473c
$s1	17	0x087f196f
$s2	18	0x00000000
$s3	19	0x00000000

Figure: After

2.5 vec_encry d, a, b

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00110000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x5d23473c
$t1	9	0x087f196f
$t2	10	0x003d0035
$t3	11	0x0011019d
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000

Figure: Before

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00110000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000011
$a1	5	0x00000889
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x5d23473c
$t1	9	0x087f196f
$t2	10	0x003d0035
$t3	11	0x0011019d
$t4	12	0x00000ca1
$t5	13	0x00000011
$t6	14	0x00000889
$t7	15	0x00000000
$s0	16	0x024d0399
$s1	17	0x029d0449
$s2	18	0x07f90932
$s3	19	0x08a30889

Figure: After

2.6 vec_or d, a, b

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x5e800000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x233c475d
$t1	9	0x087f196f
$t2	10	0x981963c5
$t3	11	0x5e80b36e
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000

Figure: Before

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x5e800000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x233c475d
$t1	9	0x087f196f
$t2	10	0x981963c5
$t3	11	0x5e80b36e
$t4	12	0x0000006f
$t5	13	0x0000006e
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0xbb3d67dd
$s1	17	0x5effbb6f
$s2	18	0x00000000
$s3	19	0x00000000

Figure: After

2.7 sort_low d, a

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x087f0000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x5d23473c
$t1	9	0x087f196f
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000

Figure: Before

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x087f0000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000001
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x5d23473c
$t1	9	0x087f196f
$t2	10	0x00000019
$t3	11	0x00000008
$t4	12	0x005d6f00
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x7f6f5d47
$s1	17	0x3c231908
$s2	18	0x00000000
$s3	19	0x00000000

Figure: After

2.8 sort_up d, a

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x087f0000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x5d23473c
$t1	9	0x087f196f
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000

Figure: Before

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x087f0000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000001
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x5d23473c
$t1	9	0x087f196f
$t2	10	0x0000006f
$t3	11	0x0000007f
$t4	12	0x00231900
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x0819233c
$s1	17	0x475d6f7f
$s2	18	0x00000000
$s3	19	0x00000000

Figure: After

2.9 subsu d, a, b

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x5e800000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x233c475d
$t1	9	0x087f196f
$t2	10	0x981963c5
$t3	11	0x5e80b36e
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000

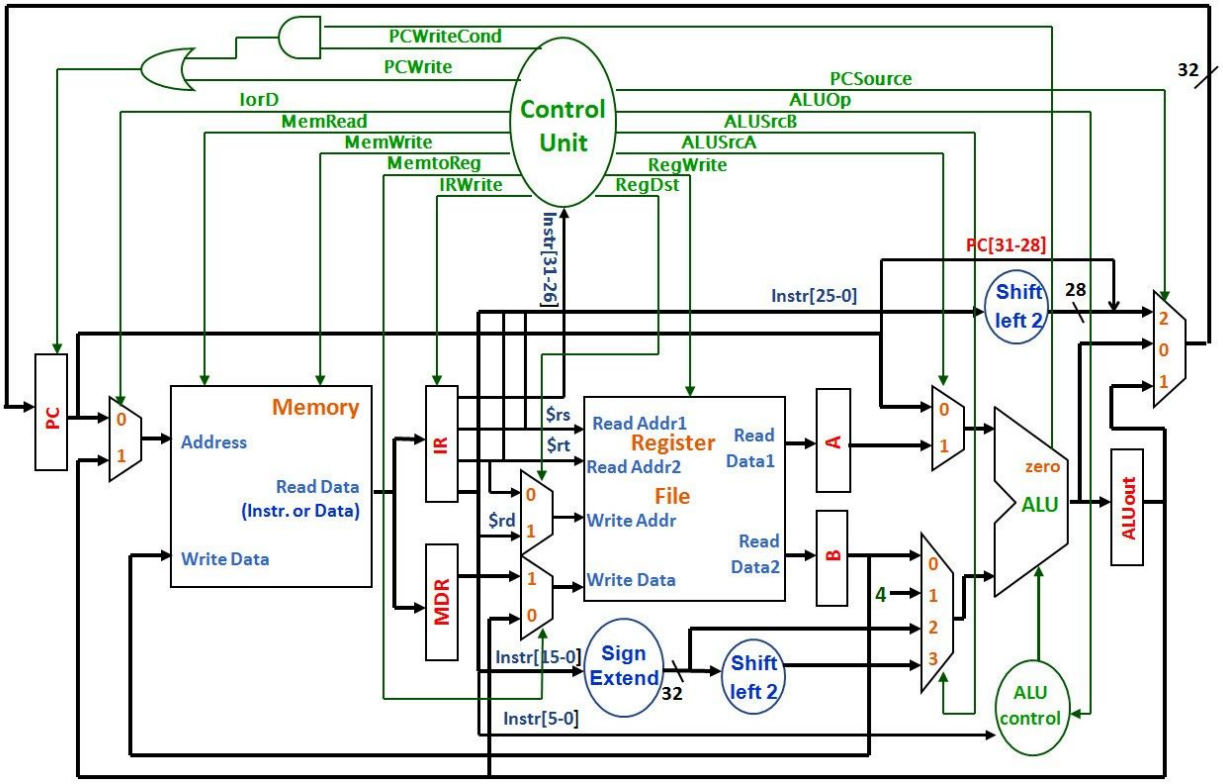
Figure: Before

Registers | Coproc 1 | Coproc 0

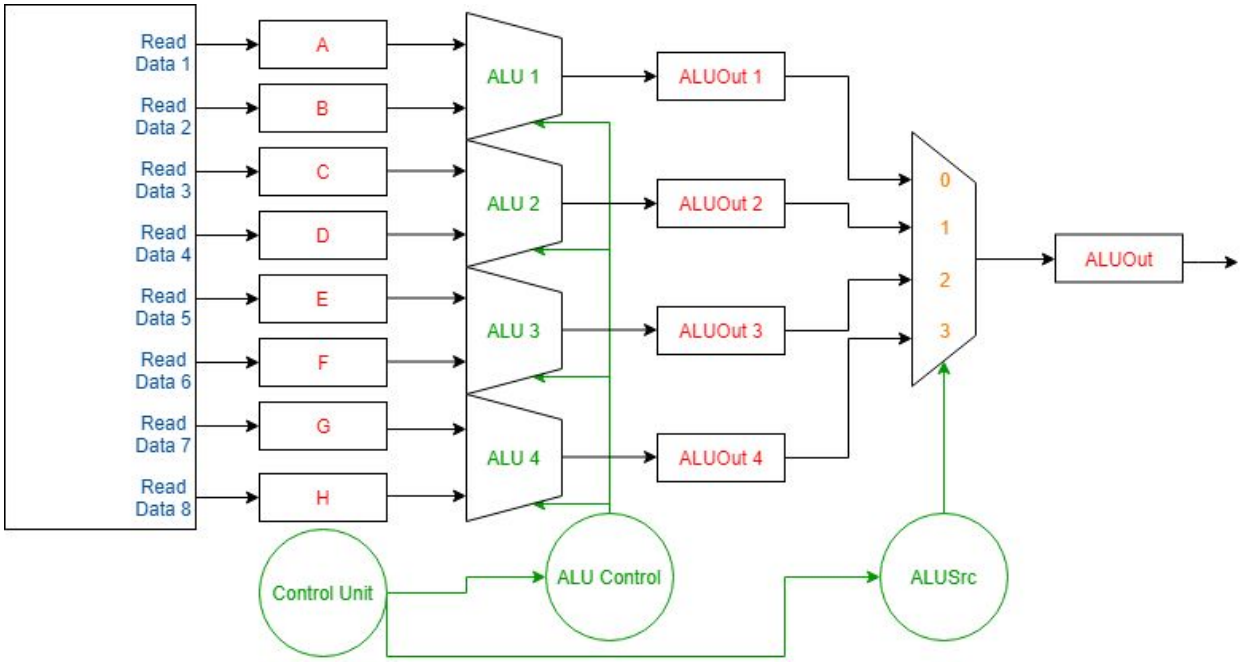
Name	Number	Value
$zero	0	0x00000000
$at	1	0x5e800000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x233c475d
$t1	9	0x087f196f
$t2	10	0x981963c5
$t3	11	0x5e80b36e
$t4	12	0x00000001
$t5	13	0x0000006e
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00230000
$s1	17	0x00000001
$s2	18	0x00000000
$s3	19	0x00000000

Figure: After

IV. Datapath Block Diagrams

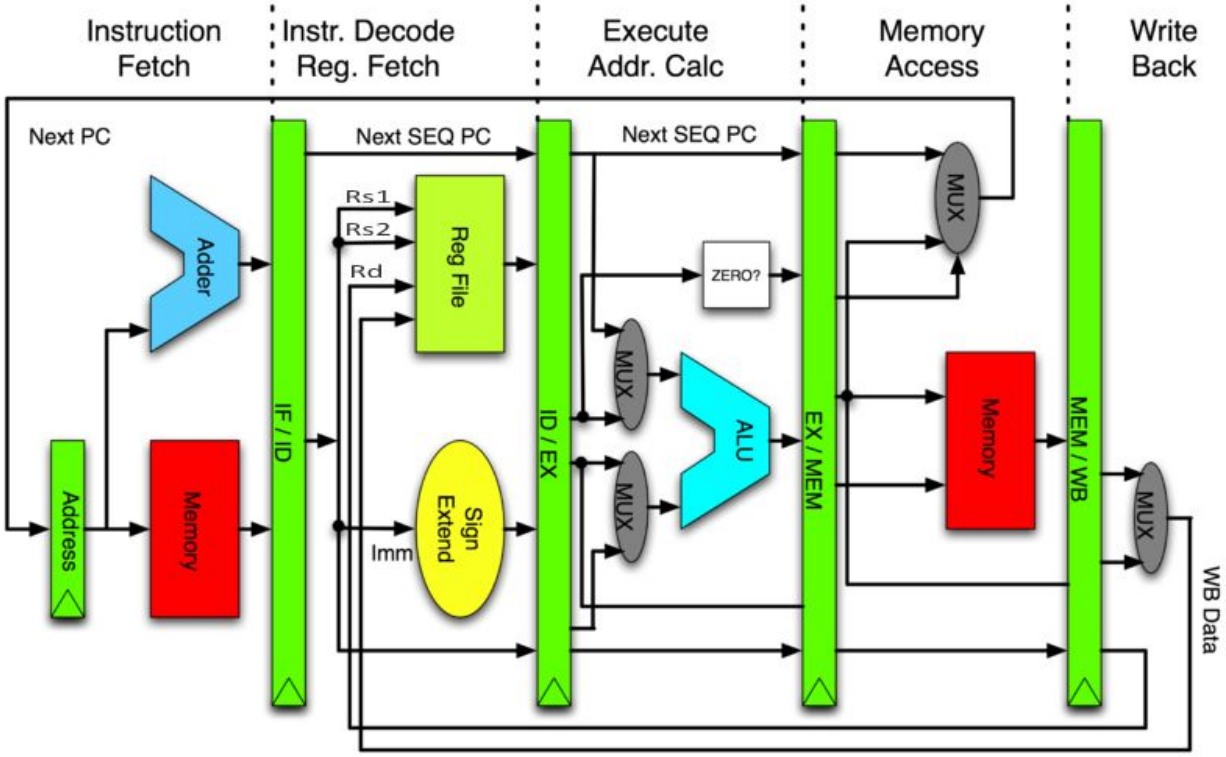


The above is the existing MIPS multi-cycle datapath. In order to effectively implement the SIMD enhancements, it must be amended.

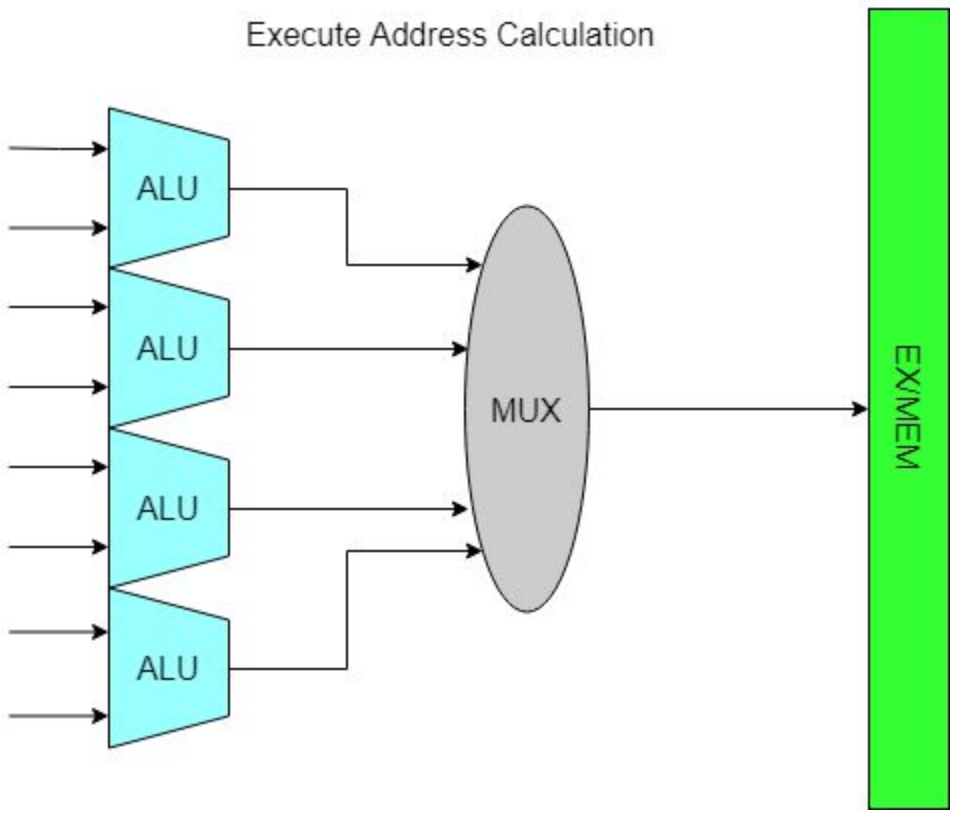


To implement, the register file and the ALU must be changed. The register file must be able to read eight values at once, referencing the eight-element vectors. An ALU must be added every two values for a total of four, allowing multiple operations to be carried out at once. ALU Control decides what operation takes place inside. The results of are stored into ALUOuts which are then connected to a MUX. A control signal decides which one is output to the main ALUOut at one time.

Any datapaths and busses not shown remain unchanged.



Above is the MIPS pipeline datapath. In much the same way as the multi-cycle datapath, it must be changed to accommodate SIMD instructions.



The changes made to the pipeline datamath mirror the ones made to the multi-cycle datapath.In the execute address calculation cycle, the ALU is changed to four different ones. The results are connected to a MUX which then decides the order in which the results move forward to the next segment.

V. Additional Comments

Creating an enhanced MIPS architecture was a challenging but enlightening process. Having done few projects of this scope before, the reference manual seemed to be a daunting task. However we did what we do for any large assignment and broke it down piece by piece, dividing responsibilities and allotting time each day to working. This was not without its challenges as we encountered multiple unexpected road bumps, the most infamous of which having to rewrite all our code from scratch due to a fatal misunderstanding error. Despite the crunch, we persevered.

The programmer reference manual was a tiring, tedious process but an ultimately rewarding one. The project mirrors the kind of tasks and challenges you would find in a real world career, and has provided invaluable experience as a result.

VI. References

The following documents are used as references (examples, formats, etc):

- Fall 2018 Semester Project - SIMD Enhanced MIPS Instructions (R.W Allison, CECS 341)
- MIPS Reference Data (R.W Allison, CECS 341)