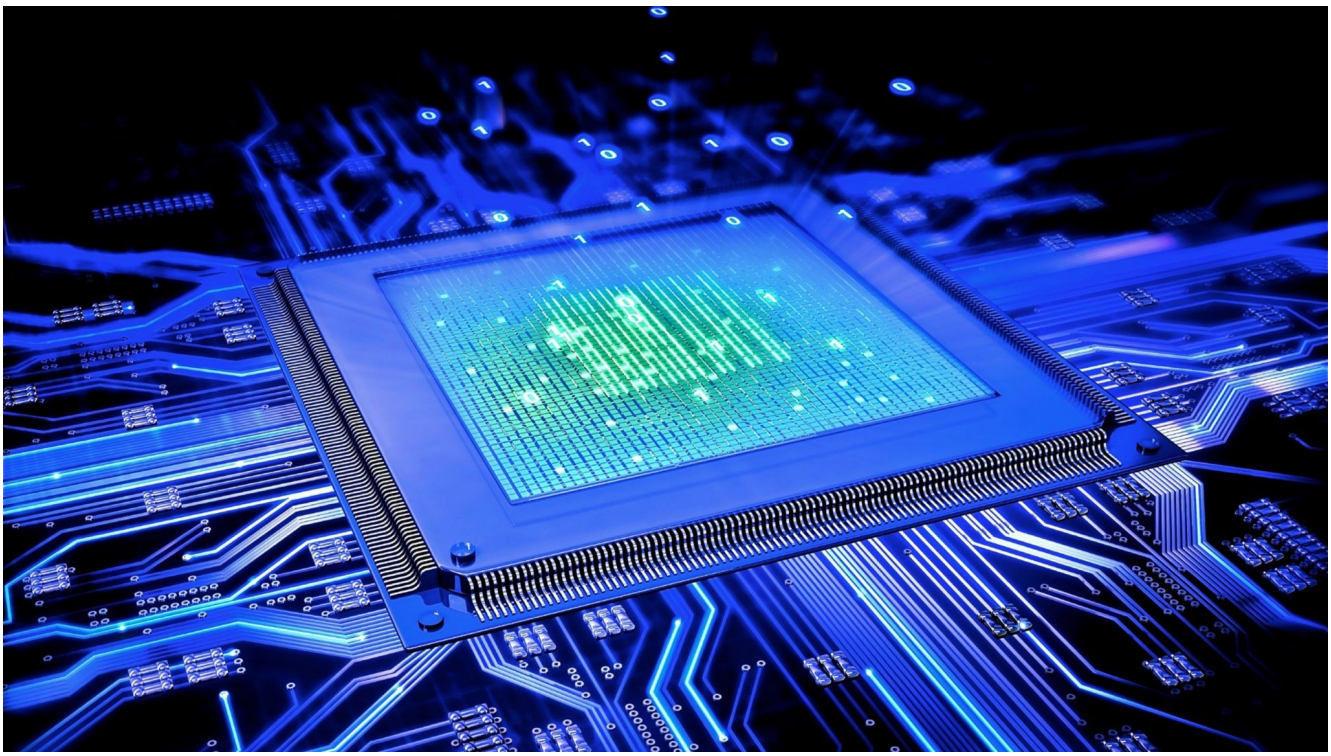
SIMD Enhanced MIPS Instructions

CECS 341 - Fall 2018

Professor Robert Allison



— Authors —

Thanh Le and Steven Chung

Table of Contents

I. Purpose

A. Abstract

At California State University Long Beach all students who wish to obtain a degree in Computer Science must take the course CECS 341. The title of CECS 341 is Computer Architecture where students learn about the MIPS processor in great detail. For our project in this course, students must group up and design their own MIPS instruction set architecture. The following document will be a reference manual to a customized MIPS ISA also called an SMID enhanced instructions. These SMID instructions are created using MIPS instructions that code new instruction formats and data types.

What is MIPS?

MIPS stands for Microprocessor without Interlocked Pipelined Stages which was first introduced in 1985. It is a RISC ISA or in other words, a Reduced Instruction Set Computer Instruction Set Architecture. MIPS is a register based architecture meaning that it uses registers with its instructions. Registers are a certain type of memory which means that the CPU uses information to store data into them. The CPU also uses these registers to read and write, the main advantage to uses registers is that they are faster than main memory. MIPS has a few different versions of its processor but the version we are going to use for the ISA we are using 32-bit registers. You may you many of these registers to your preferred liking but there are some that are reserved for special purposes. One of these registers are $Zero, the “$” represents a register, which only will hold the data zero in it. MIPS is extremely important to the creation of RISC architecture.

SMID Enhanced Instructions

SMID enhanced instructions implement and operate with vectors. Usually a SMID instruction involves two vectors and performs an operation on their elements, then stores that operation back into a vector. The main advantage of this is to have the ability for lots of data to be operated on in one single instruction. This is just one case of using SMID enhance instructions but they can be used with many different sized vectors, data types and elements. You can do so many things with SMID enhanced instructions.

B. Application

Our implementation of SIMD Enhanced Instructions include 18 instructions. Each of our instructions improves algorithms necessary for a plethora of different audiences to use them.

The Instructions:

Instruction Title	**Vector Add Saturated (unsigned)**
Function	Using two vectors. Add elements in one vector to elements in another vector and store the result into another vector.
Benefits	This instruction is very efficient and can keep older information without having to change them. If you had to calculate total amount of points scored by each player on a basketball team; you can have points scored by each player per quarter in separate vectors and add them into a separate vector holding the total points of each basketball players.
Target Audience	Programmers, data analysts, mathematicians, others

Instruction Title	**Vector Multiply and Add**
Function	Having three vectors with elements in each. Multiply two vectors' elements together and add the result with the third vector. Then store the result into a separate vector.
Benefits	This instruction will help reduce the need for too many vectors therefore simplifying methods that use multiplying and adding while also making the overall methods more efficient.
Target Audience	Programmers, data analysts, mathematicians, accountants, others

Instruction Title	**Vector Multiply Even Integer**
Function	Having vectors with multiple data in each. Only multiply the even elements of the two vectors and storing the result into a separate vector.
Benefits	An efficient way to select specific even elements from a vector multiply them together.
Target Audience	Programmers, data analysts, mathematicians, accountants, others

Instruction Title	**Vector Multiply Odd Integer**
Function	Having vectors with multiple data in each. Only multiply the odd elements of the two vectors and storing the result into a separate vector.
Benefits	An efficient way to select specific odd elements from a vector multiply them together.
Target Audience	Programmers, data analysts, mathematicians, accountants, others

Instruction Title	**Vector Multiply Sum Saturated**
Function	Multiply the elements of two vectors then saturate add (no-wrap) the product to elements in another vector. Store the result in a separate vector.
Benefits	An efficient way to add and multiply vectors together while keeping the number of vectors needed to a minimum.
Target Audience	Programmers, data analysts, game developers, mathematicians, accountants, others

Instruction Title	**Vector Splat**
Function	Copy any element from one vector into all the elements of another vector.
Benefits	An efficient way to create a new vector where all elements are the same.
Target Audience	Programmers and game developers

Instruction Title	**Vector Merge Low**
Function	Take the low elements of the first vector from left to right and place them into the even spots of a new vector. For the second vector take the low elements from left to right and place them into the odd elements of that new vector.
Benefits	An efficient way to store elements of two different vectors into one as long as losing the high elements of both vectors is okay.
Target Audience	Programmers

Instruction Title	**Vector Merge High**
Function	Take the high elements of the first vector from left to right and place them into the even spots of a new vector. For the second vector take the high elements from left to right and place them into the odd elements of that new vector.
Benefits	An efficient way to store elements of two different vectors into one as long as losing the low elements of both vectors is okay.
Target Audience	Programmers

Instruction Title	**Vector Pack**
Function	Take the truncation of the wider elements in the first vector and store it into the high elements. For the second vector we take the truncation of the wider elements and store them into the lower elements in the final vector.
Benefits	Elements from two vectors could be joined together and stored into another vector while losing the high bits of each element.
Target Audience	Programmers

Instruction Title	**Vector Permute**
Function	One vector contains an element specifier. This specifier tells where each element in each vector is supposed to go. Not every element in each vector is moved into the new vector, only ones selected by the element specifier vector.
Benefits	Elements from two vectors could be joined together and stored into another vector while losing some elements from each vector.
Target Audience	Programmers

Instruction Title	**Vector Compare Equal-To**
Function	Check if each element of two different vectors is equal to each other. In the final vector the element will be 1 or 0. 1 being TRUE and 0 being FALSE.
Benefits	This is good for checking if data is valid. Good for comparing collections of data in different vectors.
Target Audience	Programmers, software developers and data analysts

Instruction Title	**Vector Compare Less-Than**
Function	Check if each element of two different vectors is less than one another. In the final vector the element will be 1 or 0. 1 being TRUE and 0 being FALSE.
Benefits	This is good for checking if data is valid. Good for comparing collections of data in different vectors.
Target Audience	Programmers, software developers, engineers and data analysts

Instruction Title	**Vector Shift Left Arithmetic**
Function	Shift all the elements of a vector to the right and preserve the sign of vector
Benefits	This data is very useful for implementing a way to constantly adding data. It can be program to retrieve data every specified minute, second, etc.
Target Audience	Programmers, software developers, engineers and data analysts

Instruction Title	**Vector Shift Right Logically**
Function	Shift all the elements of a vector to the right. The signed bit is not preserved because it is a logical shift.
Benefits	This data is very useful for implementing a way to constantly adding data. It can be program to retrieve data every specified minute, second, etc.
Target Audience	Programmers, software developers, engineers and data analysts

Instruction Title	**Vector Shift Left Logically**
Function	Shift all the elements of a vector to the left. The signed bit is not preserved because it is a logical shift.
Benefits	This data is very useful for implementing a way to constantly adding data. It can be program to retrieve data every specified minute, second, etc.
Target Audience	Programmers, software developers, engineers and data analysts

Instruction Title	**Vector Reverse**
Function	Reverse the order of all elements in a vector
Benefits	The Vector reverse instruction will be very useful for doing the opposite of what was initially done. This is also good for flipping pictures for graphic designers.
Target Audience	Programmers, software developers, engineers and graphic designers.

Instruction Title	**Vector Replace All**
Function	Replace all elements in a vector with a specified data
Benefits	This vector will be very useful in numerous ways. One way is if a vector is made up of a collection of characters. You can replace a specific type of characters with another type of character fast and efficiently.
Target Audience	Programmers and software developers.

Instruction Title	**Vector Swap Element**
Function	Swap two elements in a vector with two indicated indexes
Benefits	This will be extremely useful because a programmer will now have the ability to swap any number of elements between two different vectors. This will save them time and is very efficient.
Target Audience	Programmers and software developers.

[1] [1] The following diagrams and visuals are taken from Robert Allison's slides or from the various websites. These are used for visuals and by no means are we attempting to claim these visuals as our own.

II. Instruction Set Architecture

A. Machine Register Set

This is the original usual machine register set that is commonly used. As you can see there are many registers that can be used but most likely the users do not need to use all of them.

NAME	NUMBER	USE
$zero	0	The constant value of 0
$at	1	Assembler Temporary
$v0 - $v1	2 - 3	Values for function results and expression evaluation
$a0 - $a3	4 - 7	Arguments
$t0 - $t7	8 - 15	Temporaries
$s0 - $s7	16 - 23	Saved Temporaries
$t8 - $t9	24 - 25	Temporaries
$k0 - $k1	26 - 27	Reserved for OS Kernel
$gp	28	Global Pointer
$sp	29	Stack Pointer
$fp	30	Frame Pointer
$sra	31	Return Address

For this SIMD enhanced set of registers we decided to simplify the Instruction Set Architecture by reducing the amount of registers available to the user. The following registers are the only registers available to the users in our version of SIMD Enhanced Set of Registers ISA.

B. Data Types

Data types are extremely important to know before working with code. Our specific SIMD enhanced architecture uses only a limited amount of data types to make the coding experience a lot simpler for the user.

Byte

The Byte data type is used for single integers without any decimal places. Not only can it store integers but this data type can also store characters.

Word

The Word data type can be used to initialize an array or store data. It is in the form of 32 bits.

Float

The float data type is used to store an integer value with a single decimal value.

Double

The double data type is used to store double value with more than one decimal place.

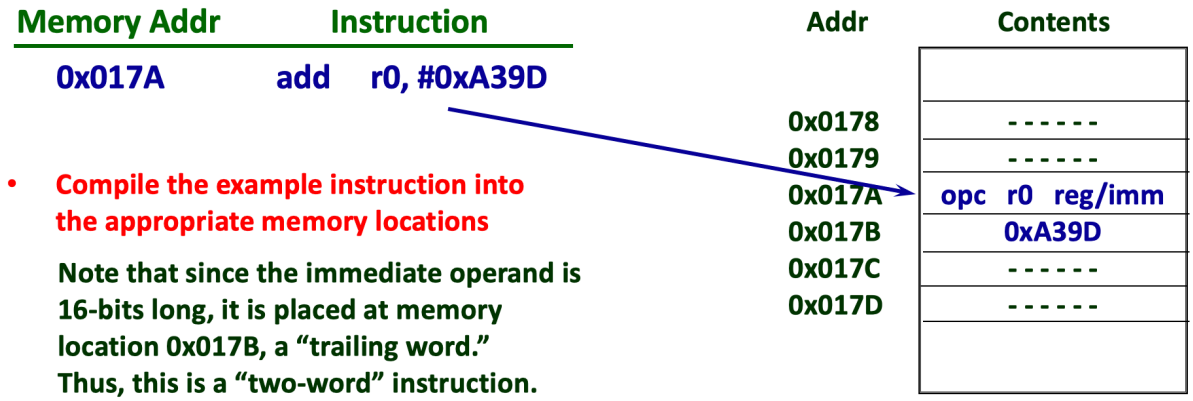
C. Addressing Modes

An addressing mode is the way an operand of an instruction is specified. They specify the rules for modifying the address fields of the instructions prior to being executed. In other words they specify an operand or a memory address at run time.

The Five Ways of Addressing Data:

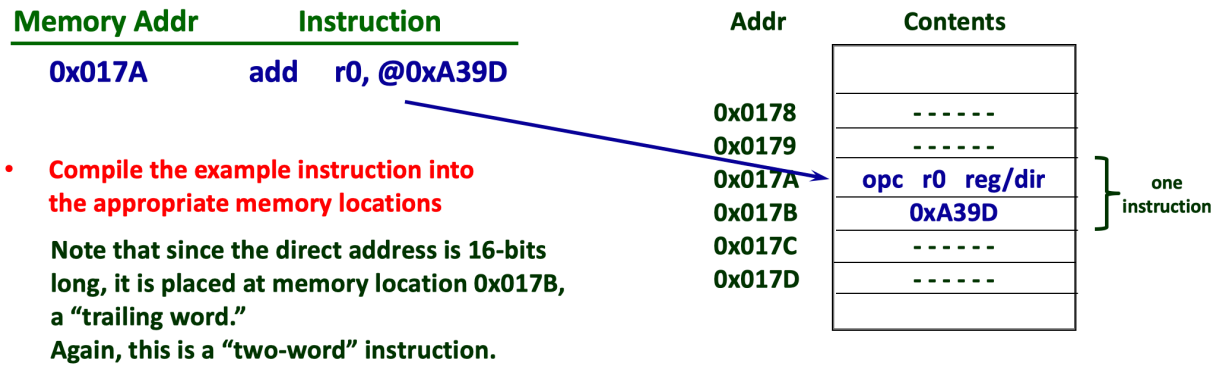
1. Immediate

The operand itself is part of the instruction. This mode is effective when initializing registers with address or data constants.



2. Direct

The instruction contains the address of the operand. Direct addressing may also be referred to as “absolute” addressing. The address must be know at compile time.

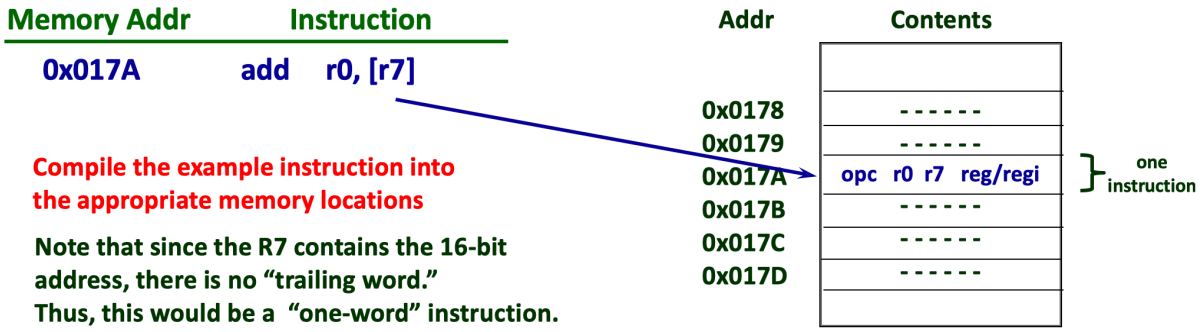


- Compile the example instruction into the appropriate memory locations

  Note that since the direct address is 16-bits long, it is placed at memory location 0x017B, a “trailing word.”
  Again, this is a “two-word” instruction.

3. Register Indirect

The instruction specifies the register holding the address of the operand. This addressing mode us used to access memory operands pointed to by “base pointers” that are stored in registers.

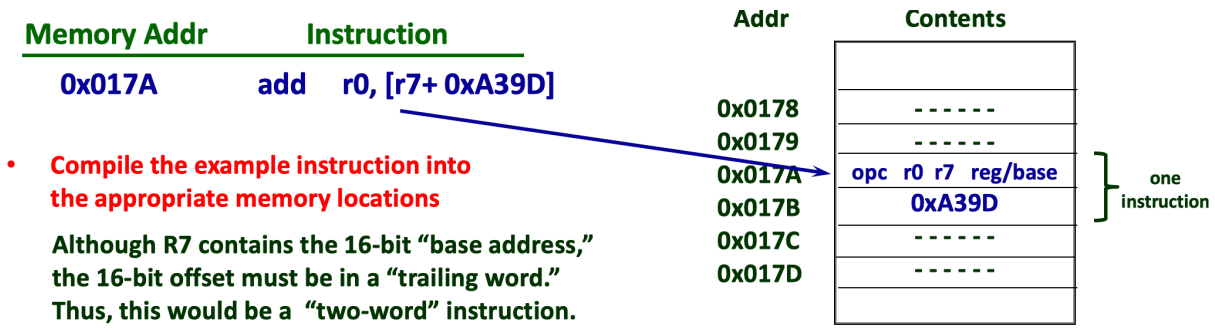


- Compile the example instruction into the appropriate memory locations

  Note that since the R7 contains the 16-bit address, there is no “trailing word.” Thus, this would be a “one-word” instruction.

4. Base (Indexed)

A variation of register indirect mode, where the effective address of the operand is calculated by the addition of a “base” register and an offset (signed displacement) included as a field in the instruction. This mode can also be known as “Register Indirect with Offset.”

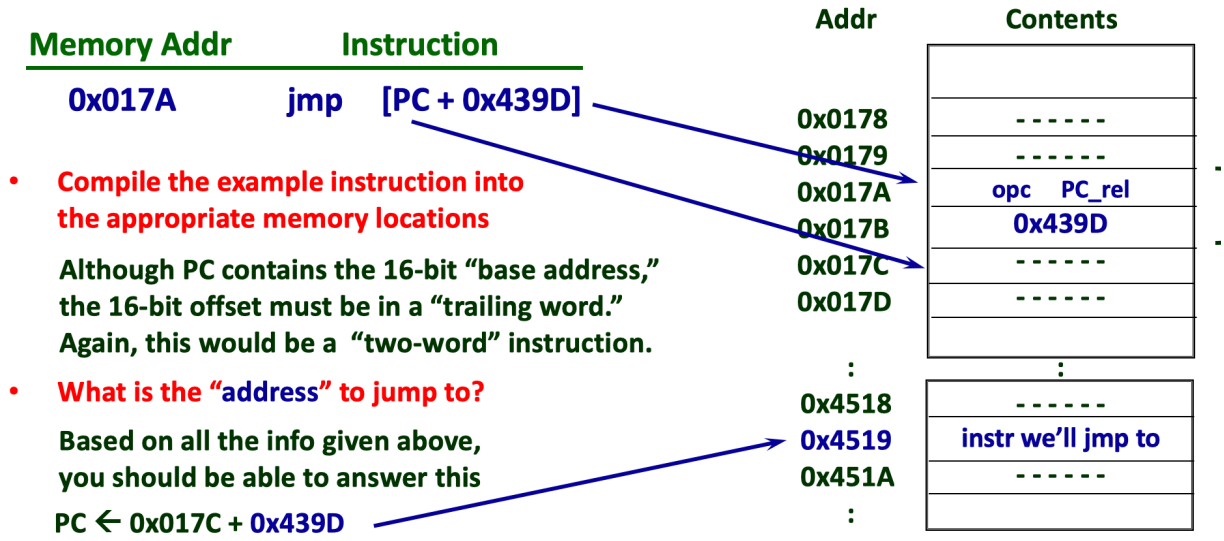


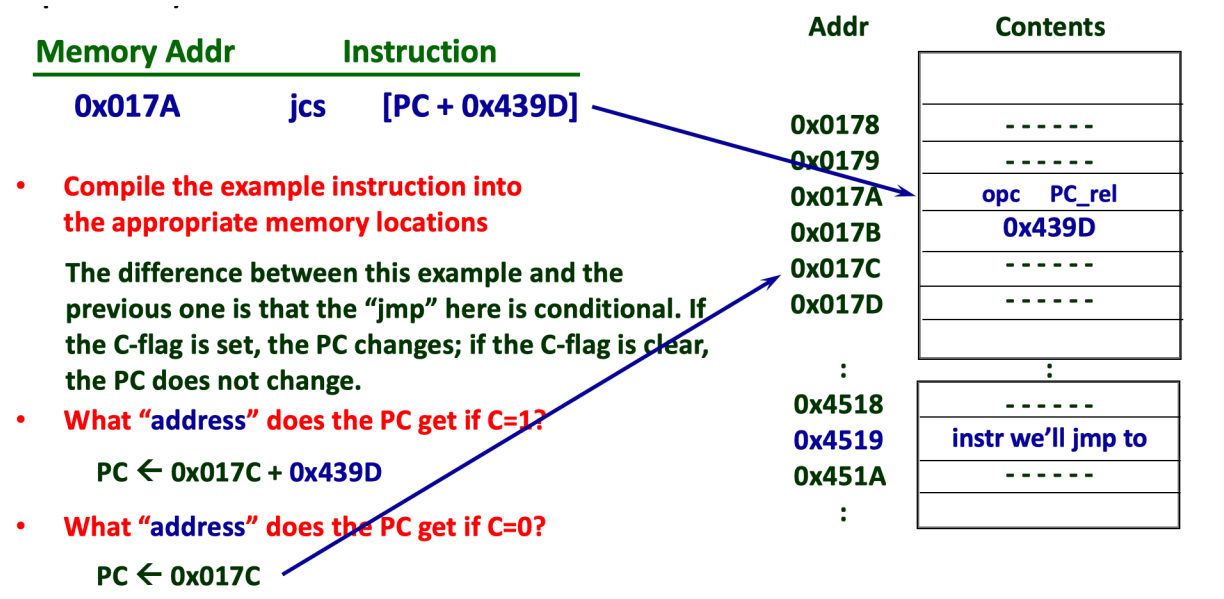
- Compile the example instruction into the appropriate memory locations

  Although R7 contains the 16-bit “base address,” the 16-bit offset must be in a “trailing word.” Thus, this would be a “two-word” instruction.

5. PC Relative

A variation of “base mode”, where the effective address of the operand (typically an address to jump to) is calculated by the addition of the current PC and an offset (signed displacement) included as a field in the instruction.





D. Instruction Set and Binary Instruction Formats

The Three Basic Instruction Formats

R-Type Instructions

R-type Instructions are used when all the data values used by instructions are located in registers.

Format:

OP rd, rs, rt

Opcode	Rs	Rt	Rd	shift (shamt)	Funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Opcode:

The opcode is the machine code that represents the instruction mnemonic.

rs, rt, rd:

Numeric representation of the source and destination registers

Shift (shamt):

The amount by which the source operand (rs) is shifted by.

Funct:

This is for instructions that share an opcode. It is used to tell the difference between the instructions with the same opcode.

I-Type Instructions

I-Type Instructions are used when the instructions needs to be operate on an immediate value and register value.

Format:

- general: OP rt, IMM(rs)
- beq and bne: OP rs, rt, IMM

Opcode	Rs	Rt	IMM
6 bits	5 bits	5 bits	16 bits

Opcode:

In all I-Type Instructions, all mnemonic belong to one and only one opcode because there is no funct parameter.

rs, rt:

rs is the source register operand and rt is the target register operand.

IMM:

The 16-bit immediate value that is used to offset values in various instructions.

J-Type Instructions

A J-type Instructions is used when a jump needs to be performed.

Format:

OP Label

Opcode	Pseudo-Address
6 bits	26-bits

Opcode:

The opcode corresponds to a certain jump command.

Pseudo-Address:

This is the address of the destination that is specified to be jumped to.

1. Triple Operand Instructions

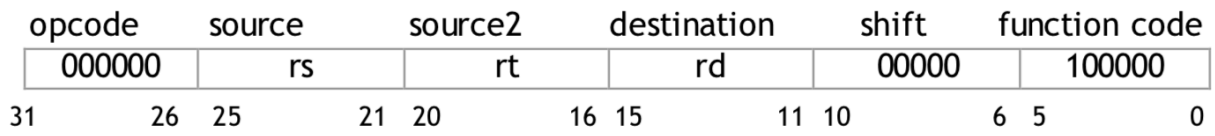
Add (add)

Description:

The add instruction takes a value from two different registers, adds them together and stores the sum into a specified register.

Binary Instruction Format:

R-Type



Mnemonic:

add rd, rs, rt

Operation:

R[rd] ← R[rs] + R[rt]

Example:

$t2 = 4, $t0 = 11

add $t1, $t2, $t0

$t1 = 15

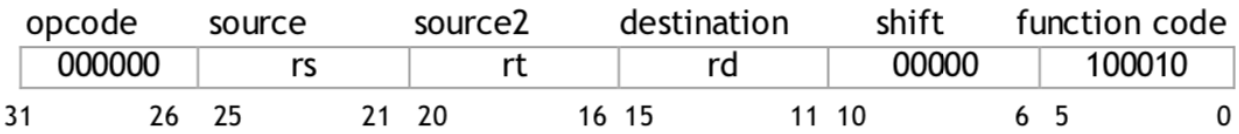
Subtract (sub)

Description:

The sub instruction takes a value from two different registers and subtracts them. The difference is then stored in the specified register.

Binary Instruction Format:

R-Type



Mnemonic:

sub rd, rs, rt

Operation:

R[rd] ← R[rs] - R[rt]

Example:

$t2 = 8, $t0 = 5

sub $t1, $t2, $t0

$t1 = 3

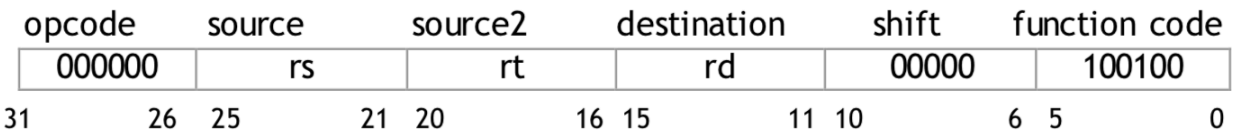
And (and)

Description:

The and instruction bitwise ands the values from two different registers and stores the result in the specified register.

Binary Instruction Format:

R-Type



Mnemonic:

And rd, rs, rt

Operation:

R[rd] ← R[rs] & R[rt]

Example:

$t1 = 1210C, $t2 = 3411B

and $t0, $t1, $t2

$t0 = 10108

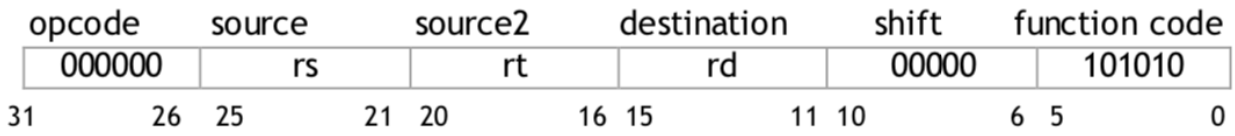
Set Less Than (slt)

Description:

The Set Less Than instruction will set the specified destination register equal to 1 if the first register's data is less than the second register's data.

Binary Instruction Format:

R-Type



Mnemonic:

slt rd, rs, rt

Operation:

R[rd] ← R[rs] & R[rt]

Example:

$t1 = 1, $t2 = 8

and $t0, $t1, $t2

$t0 = 1

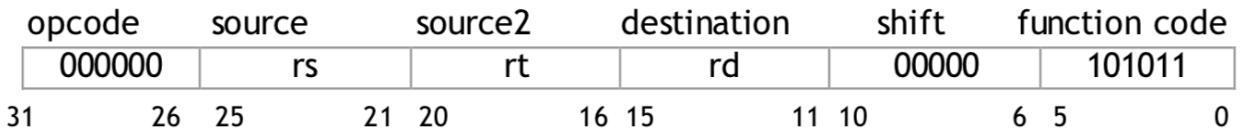
Set Less Than Unsigned (sltu)

Description:

The set less than unsigned instruction will see the destination register equal to one only if the unsigned value that is stored in the first register is less than the unsigned value stored in the second register.

Binary Instruction Format:

R-Type



Mnemonic:

sltu rd, rs, rt

Operation:

R[rd] ← (R[rs] < R[rt])

Example:

$t1 = 25 $t0 = 3
sltu $t2, $t1, $t0
After this instruction is executed $t2 would have the value 1.

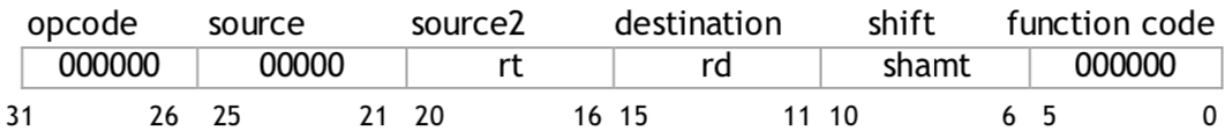
Shift Left Logical (sll)

Description:

Shifts the register operand left by the 5-bit immediate value and stores the result in the specified register.

Binary Instruction Format:

R-Type



Mnemonic:

sll rd, rt, imm5

Operation:

R[rd] ← R[rt] << imm5

Example:

$t1 = 0x6F, imm5 = 2

and $t0, $t1, 2

$t0 = 0xBC

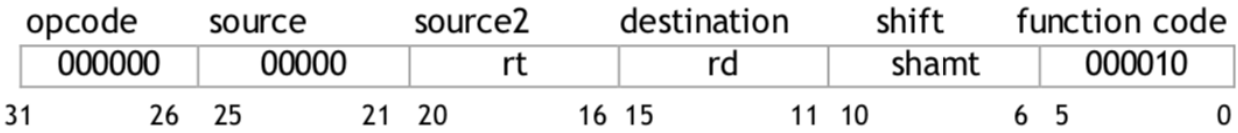
Shift Right Logical (srl)

Description:

Shifts the register operand right by the 5-bit immediate value and stores the result in the specified register.

Binary Instruction Format:

R-type



Mnemonic:

srl rd, rt, imm5

Operation:

R[rd] ← R[rt] >> imm5

Example:

$t1 = 0xBC, imm5 = 2

and $t0, $t1, 2

$t0 = 0x6F

2. Double Operand Instructions

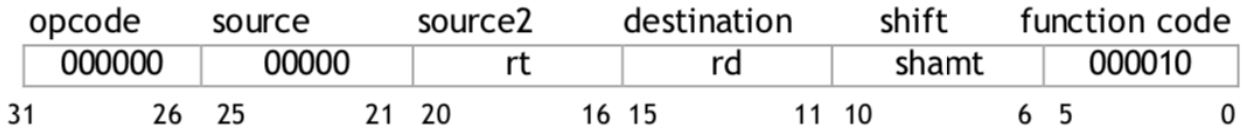
Multiply (mult)

Description:

The Multiply instruction takes values from two different registers and store the product in Lo and Hi. In the 32-bit binary instruction format, the upper 31-bits are stored in Hi and the lower 32-bits are sorted in Lo.

Binary Instruction Format:

R-Type



Mnemonic:

mult rs, rt

Operation:

Lo ← R[rs] * R[rt](31:0), Hi ← R[rs] * R[rt](63:32)

Example:

$t1 = -5, $t0 = 2
mult $t1, $t0
Lo = 0xFFFFFFF6
Hi = 0xFFFFFFFF

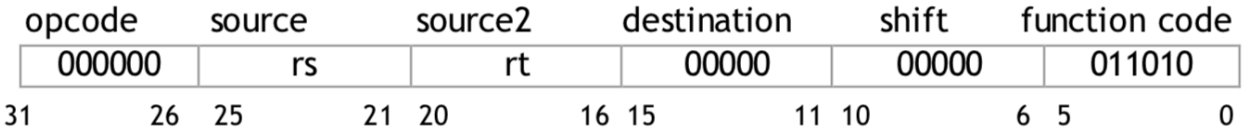
Divide (div)

Description:

The divide instruction divides the values stored in two registers and store the quotient in Lo and Hi. In the 32-bit binary instruction format, the upper 31-bits are stored in Hi and the lower 32-bits are sorted in Lo.

Binary Instruction Format:

R-Type



Mnemonic:

div rs, rt

Operation:

Lo ← R[rs] / R[rt], Hi ← R[rs] % R[rt]

Example:

$t1 = 25, $t0 = -6

div $t2, $t1

Lo = -4 and Hi = 1

3. Single Operand Instructions

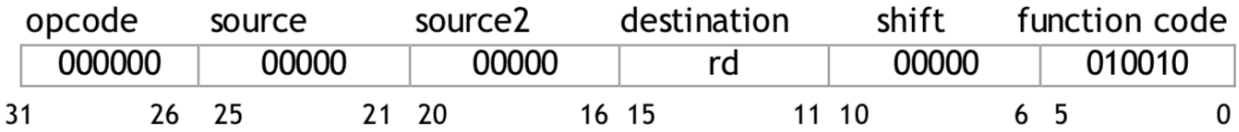
Move From Lo (mflo)

Description:

The Move From Lo instruction moves the value that is in the Lo register into the operand register.

Binary Instruction Format:

R-Type



Mnemonic:

mflo rd

Operation:

R[rd] ← Lo

Example:

Lo = 8

mflo $t1

After this instruction executes, $t1 = 8

4. Conditional Branches

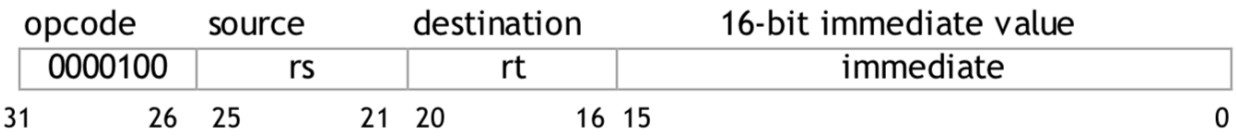
Branch On Not Equal (bne)

Description:

The Branch On Not Equal instruction branches to a specified destination if the values of two different registers are not equal to each other.

Binary Instruction Format:

I-Type



Mnemonic:

bne rs, rt, Label

Operation:

```
if (rs != rt)
PC ← PC + 4 + imm16
else
PC ← PC + 4
```

Example:

$t1 = 2, $t0 = 4
bne $t1, $t0, Label
Since $t1 and $t0 are not equal then the instruction will jump to the Label.

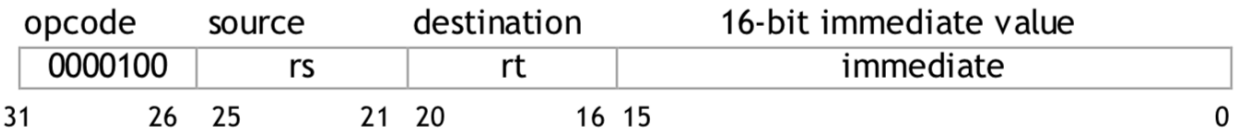
Branch On Equal (beq)

Description:

If the values that are stored in the register operands are equal, the PC gets set equal to PC + 16-bit immediate value.

Binary Instruction Format:

I-Type



Mnemonic:

beq rs, rt, Label

Operation:

```
if (rs == rt)
PC ← PC + 4 + imm16
else
PC ← PC + 4
```

Example:

beq $t1, $t0, Label

If $t1 is equal to $t0 then the PC will be branched to the Label location. If not the instruction will just continue.

5. Unconditional Jump and Subroutine Call/Return Instructions

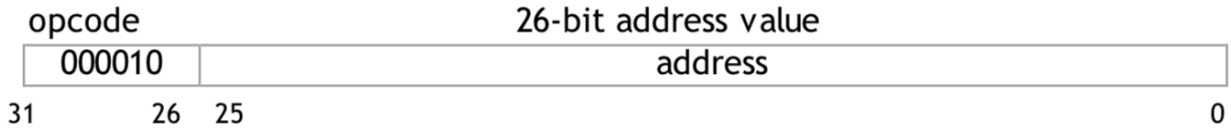
Jump (j)

Description:

This instruction jumps to the address specified by the 26-bit immediate value which is represented by a specified destination or a label.

Binary Instruction Format:

J-Type



Mnemonic:

j Label

Operation:

PC ← imm26

Example:

j Label

Sets the PC equal to whatever the value that Label contains.

6. Immediate Operand Instructions

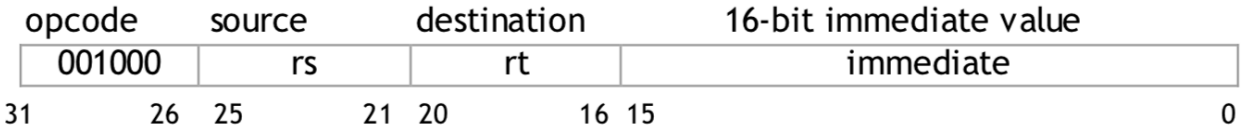
Set Less Than Immediate (slti)

Description:

If the value in the register is less than the signed 16-bit immediate. The destination register will be set to one.

Binary Instruction Format:

I-Type



Mnemonic:

slti rt, rs, imm16

Operation:

$$R[rt] \leftarrow R[rs] < imm16$$

Example:

$t1 = 3
$t0, $t1, -2

Since $t1 = 3, which is greater than -2;
$t0 will be set to 0.

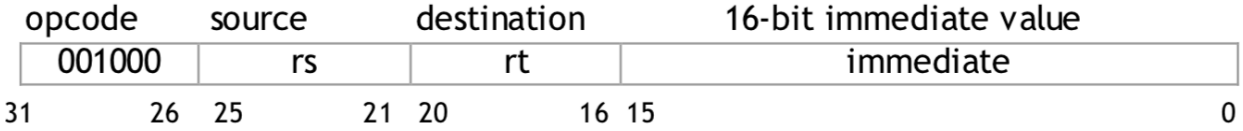
Add Immediate (addi)

Description:

The add immediate instruction adds the value of the register with a 16-bit sign immediate extended value and store the sum of into the specified register.

Binary Instruction Format:

I-Type



Mnemonic:

addi rt, rs, imm16

Operation:

$$R[rt] \leftarrow R[rs] + imm16$$

Example:

$t0 = 16

addi $t1, $t0, 2

$t1 = 18

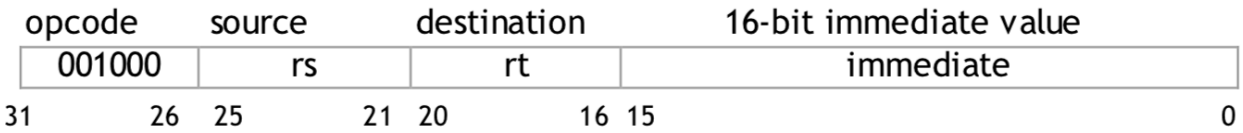
Load Immediate (li)

Description:

The load immediate instruction loads a register with a value that is immediately available. It does this all without access memory.

Binary Instruction Format:

I-Type



Mnemonic:

li rt, imm16

Operation:

R[rt] ← imm16

Example:

li $t1, 12

$t1 = 12

E. Summary of The Instruction Formats of Enhancements

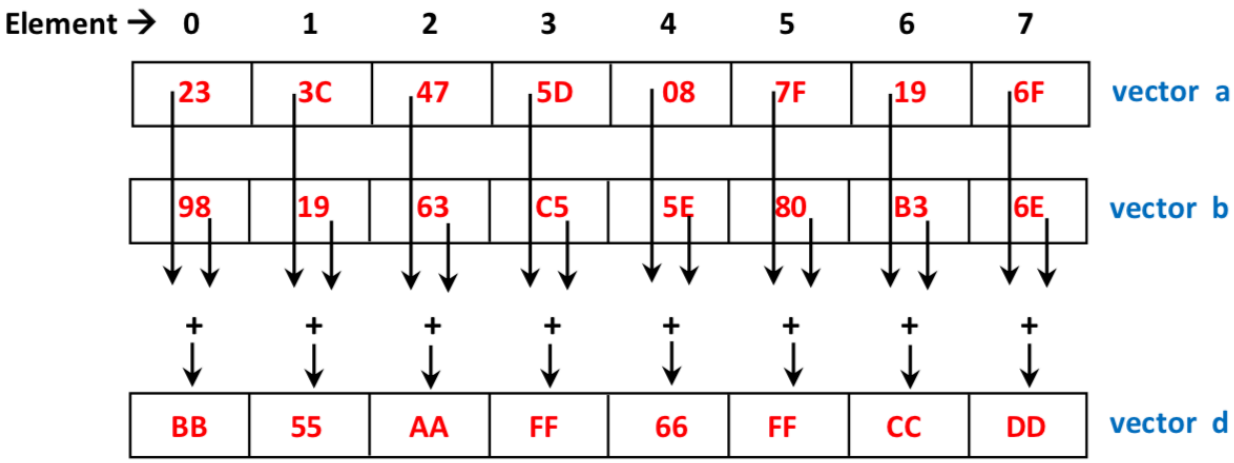
Vector Add Saturated (unsigned)

Syntax: vec_addsu $vector d, $vector a, $vector b

Description:

Each element of a is added to the corresponding element of b. The unsigned-integer (no-wrap) is placed into the corresponding element of d.

Image:



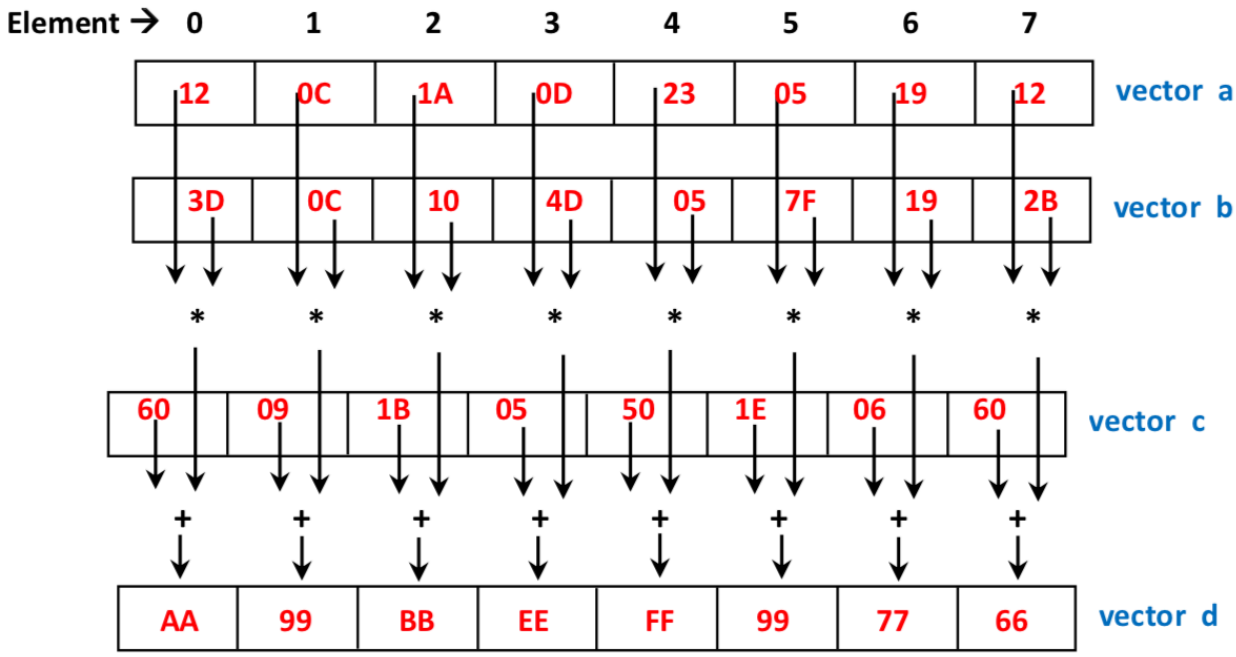
Vector Multiply and Add

Syntax: vec_madd $vector d, $vector a, $vector b, $vector c

Description:

Each elements in a is multiplied by each element in b. The intermediate result is add to the corresponding element in c, and the final sum is stored in d after being "truncated" for a half-length.

Image:



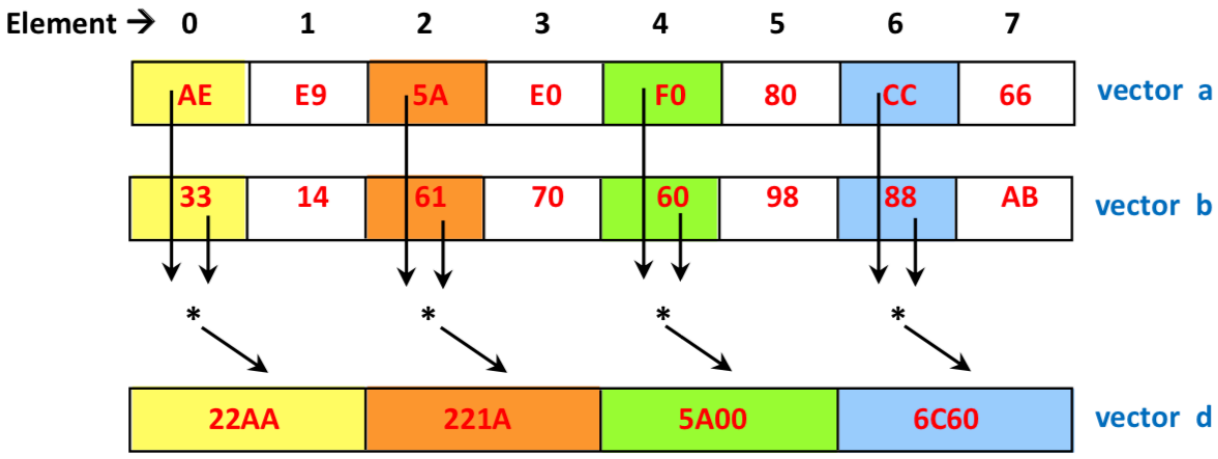
Vector Multiply Even Integer

Syntax: vec_mule $vector d, $vector a, $vector b

Description:

Each even element of a is added to the corresponding even element of b. The result is stored in full-length (16-bit) in each element of vector d.

Image:



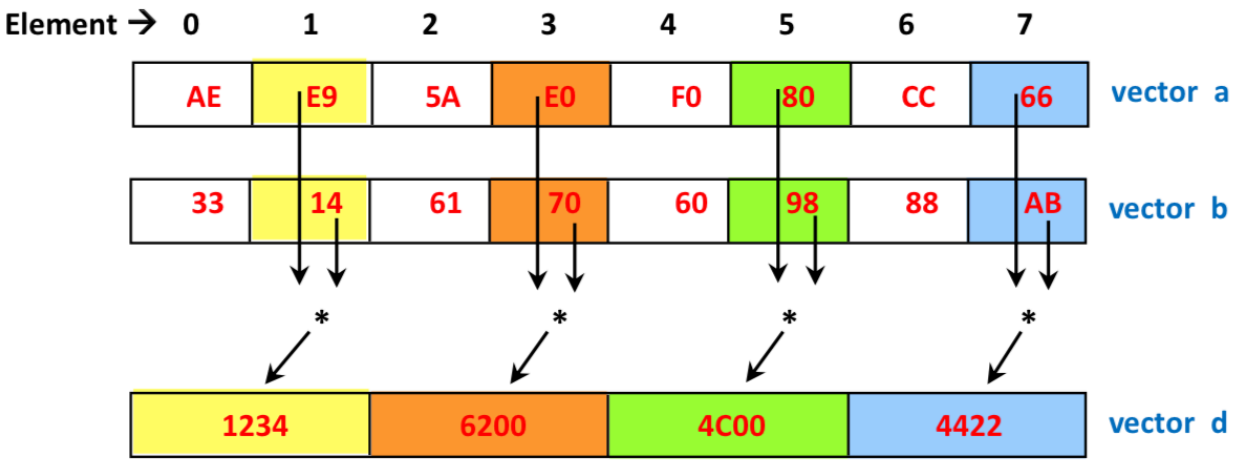
Vector Multiply Odd Integer

Syntax: vec_mulo $vector d, $vector a, $vector b

Description:

Each even element of a is added to the corresponding even element of b. The result is stored in full-length (16-bit) in each element of vector d.

Image:



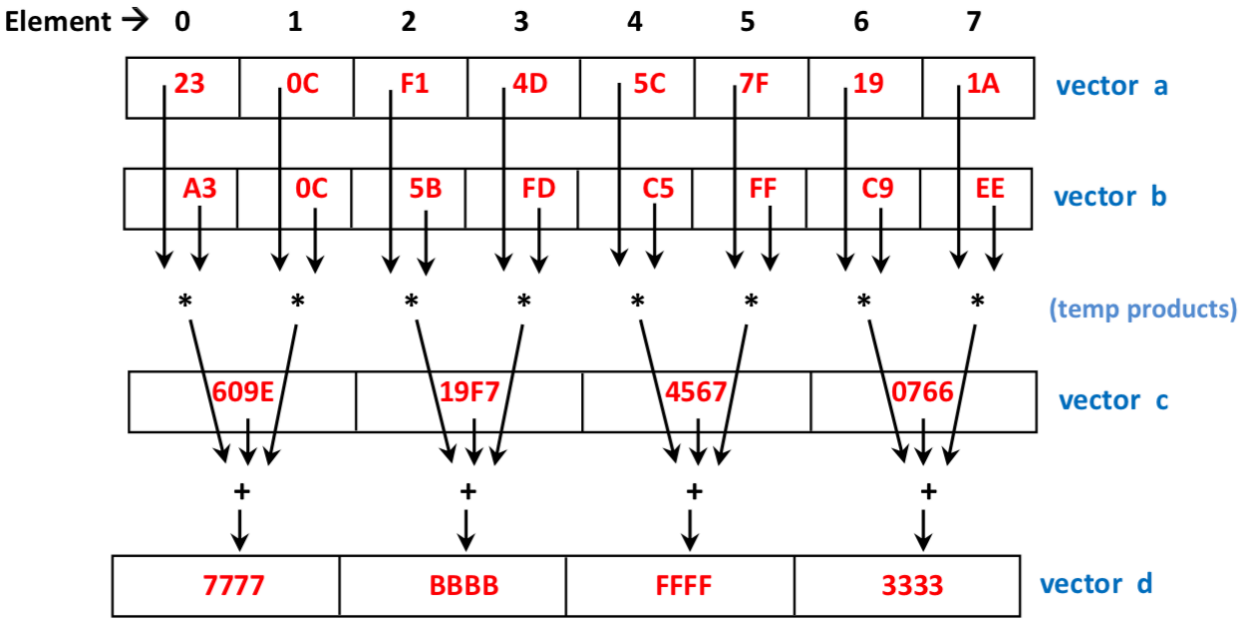
Vector Multiply Sum Saturated

Syntax: vec_msums $vector d, $vector a, $vector b, $vector c

Description:

Each element of vector d is the 16-bit sum of the corresponding elements of vector c and the 16-bit “temp" products of the 8-bit elements of vector a and vector b which overlap the positions of that element in c. The sum is performed with 16-bit saturating addition (no-wrap)

Image:



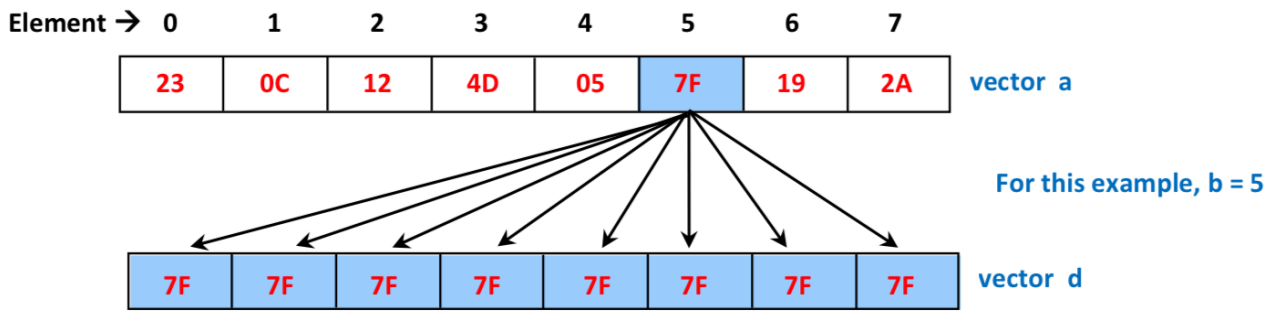
Vector Splat

Syntax: vec_splat $vector d, $vector a, $vector b

Description:

Copies any element which is indicated by component b from vector a into all of the elements of vector d.

Image:



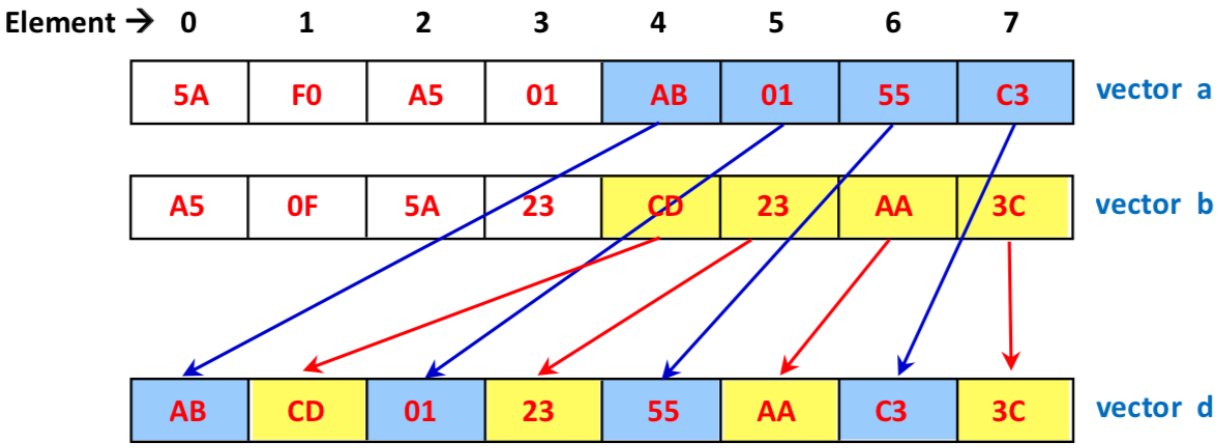
Vector Merge Low

Syntax: vec_mergel $vector d, $vector a, $vector b

Description:

The even elements of the result vector d are obtained left-to-right from the low elements of vector a. The odd elements of the result are obtained left-to-right from the low elements of vector b.

Image:



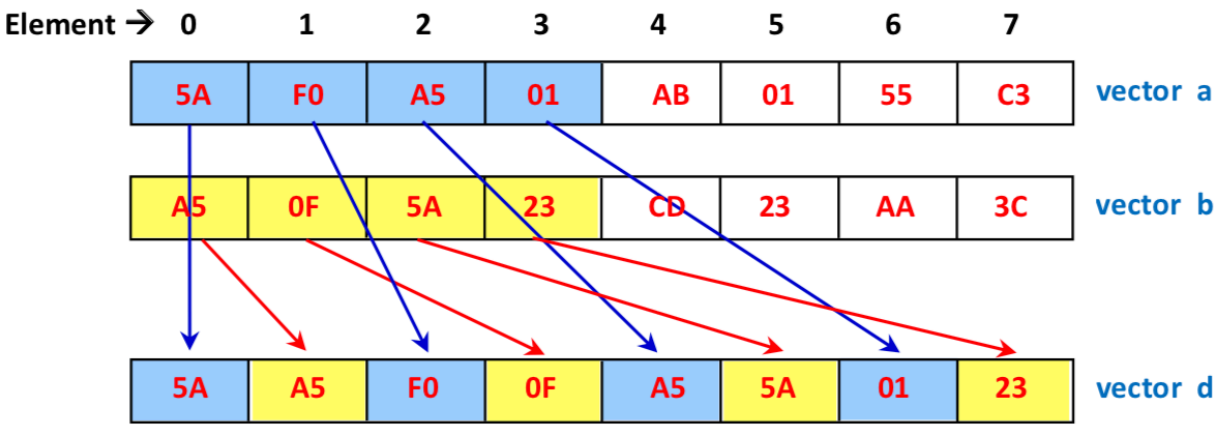
Vector Merge High

Syntax: vec_mergeh $vector d, $vector a, $vector b

Description:

The even elements of the result vector d are obtained left-to-right from the high elements of vector a. The odd elements of the result are obtained left-to-right from the high elements of vector b.

Image:



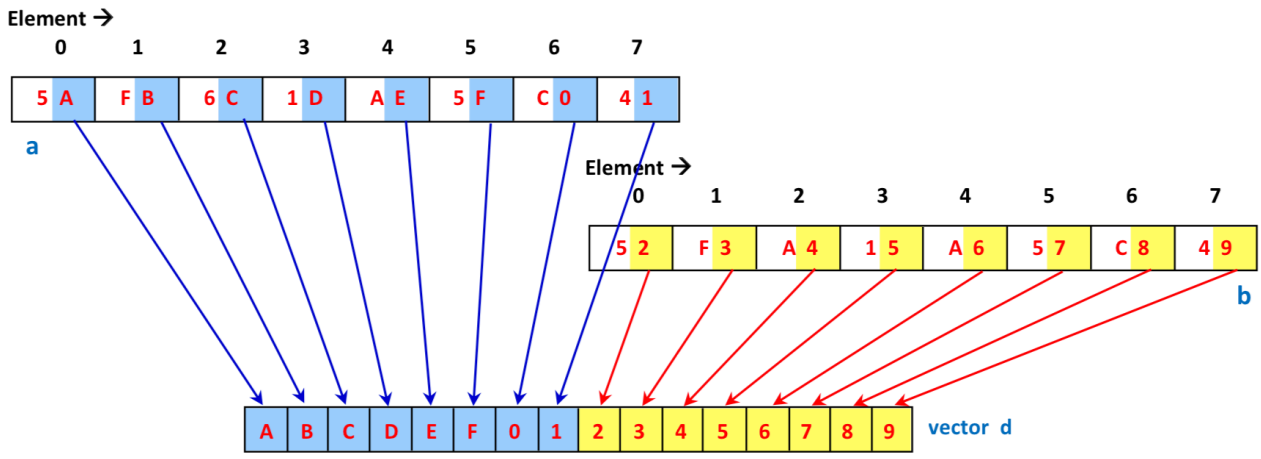
Vector Pack

Syntax: vec_pack $vector d, $vector a, $vector b

Description:

Each high element of the vector d is the truncation of the corresponding wider element of a, and each low element of the vector d is the truncation of the corresponding wider element of b.

Image:



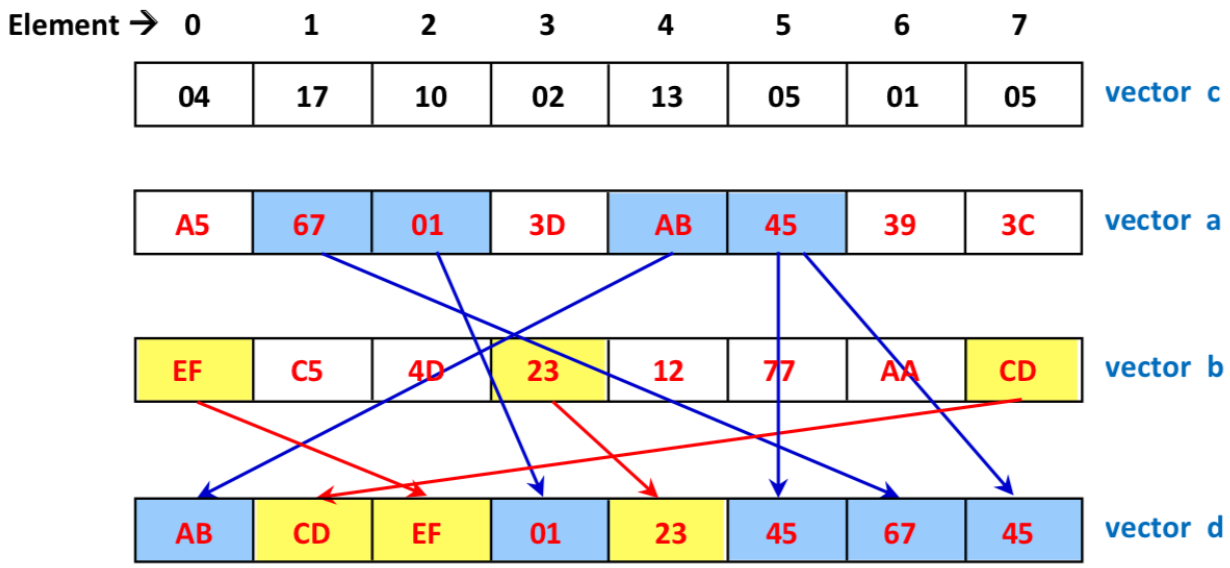
Vector Permute

Syntax: vec_perm $vector d, $vector a, $vector b, $vector c

Description:

Fills the result vector d with elements from either vector a or vector b, depending upon the "element specifier" in vector c. Each "element specifier" has two components: the most-significant half specifies an element from vector a or b (0 = a, 1 = b); the least-significant half specifies which element within the selected vector (0..7)

Image:



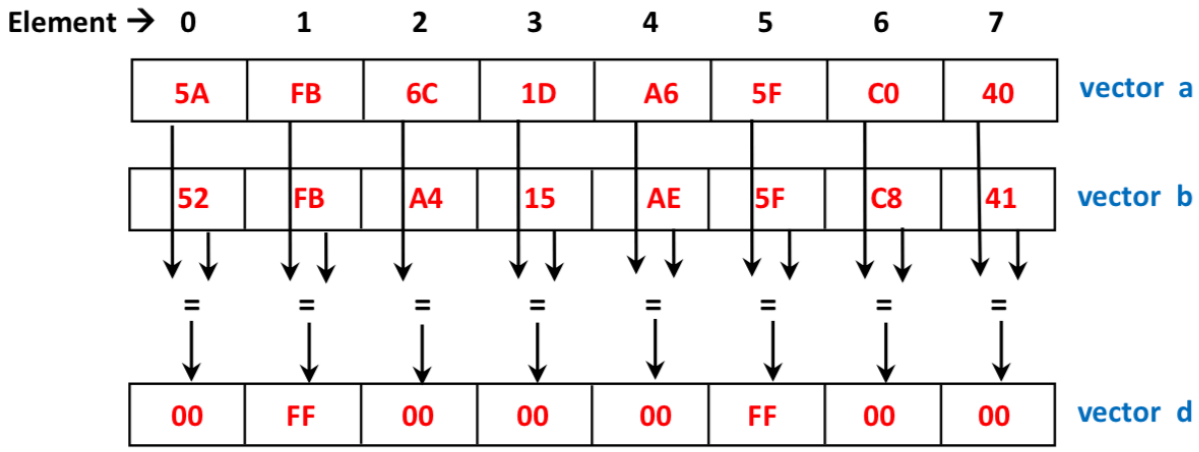
Vector Compare Equal-To

Syntax: vec_cmpeq $vector d, $vector a, $vector b

Description:

Each element of the result vector d is TRUE (all bits = 1) if the corresponding element of vector a is equal to the corresponding element of vector b. Otherwise, the element of result is FALSE (all bits = 0)

Image:



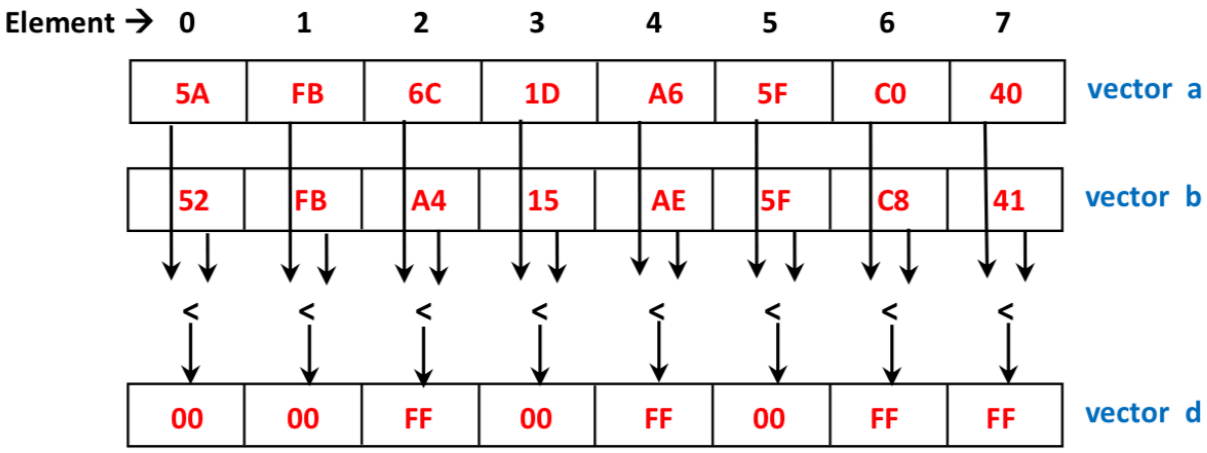
Vector Compare Less-Than

Syntax: vec_cmpltu $vector d, $vector a, $vector b

Description:

Each element of the result vector d is TRUE (all bits = 1) if the corresponding element of vector a is less than the corresponding element of vector b. Otherwise, the element of result is FALSE (all bits = 0)

Image:



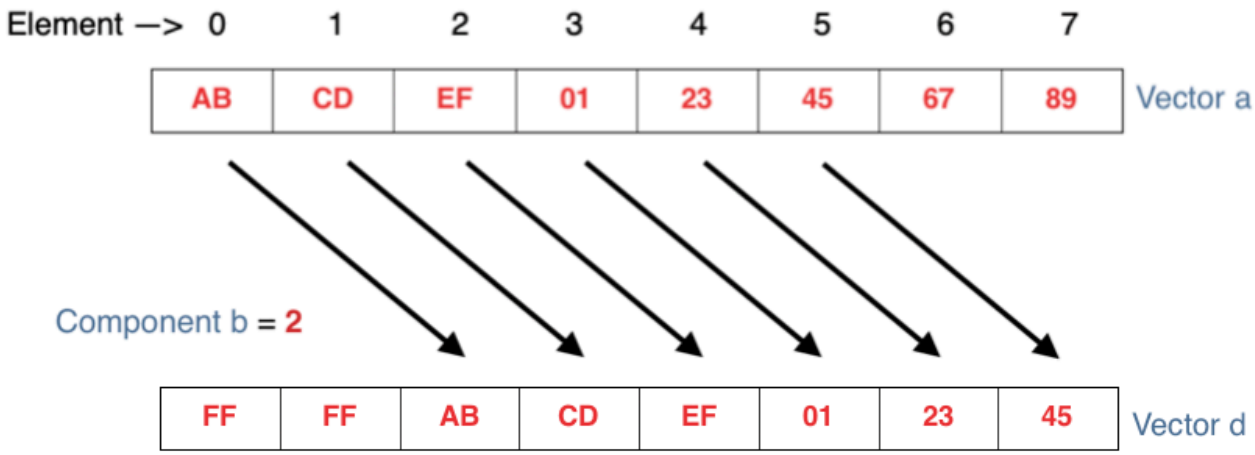
Vector Shift Right Arithmetically

Syntax: vec_sra $vector d, $vector a, $component b

Description:

Shifts vector a to the right by the number of elements indicated in component b and stores the result in vector d. Vector d has the same sign with vector a.

Image:



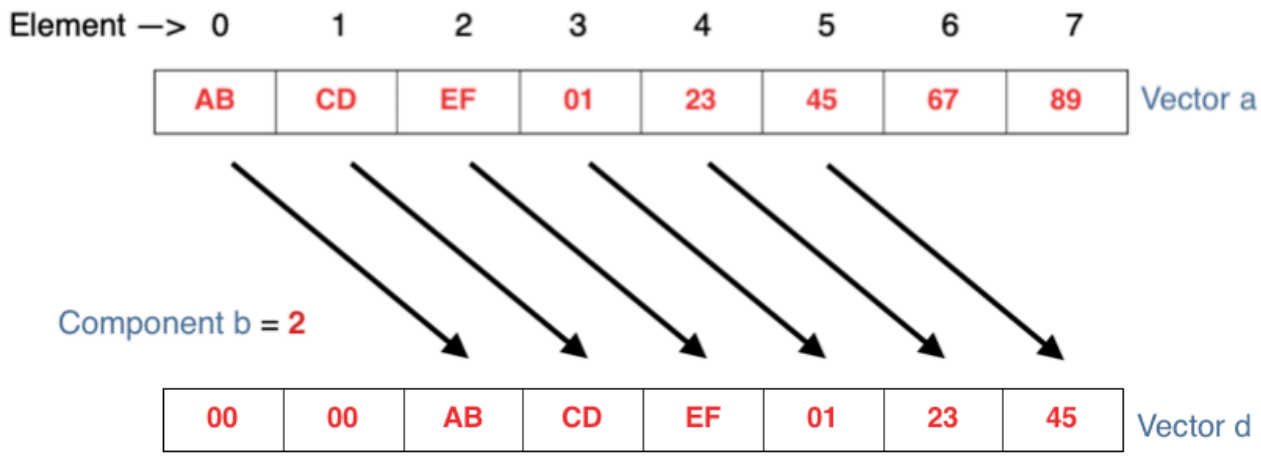
Vector Shift Left Logically

Syntax: vec_sll $vector d, $vector a, $component b

Description:

Shifts vector a to the left by the number of elements indicated in component b and stores the result in vector d.

Image:



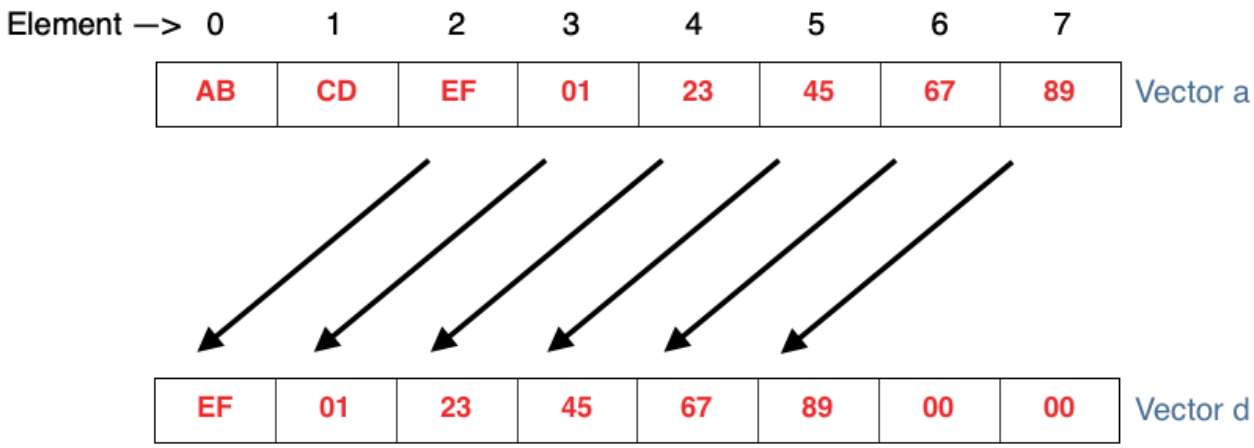
Vector Shift Right Logically

Syntax: vec_srl $vector d, $vector a, $component b

Description:

Shifts vector a to the right by the number of elements indicated in component b and stores the result in vector d.

Image:



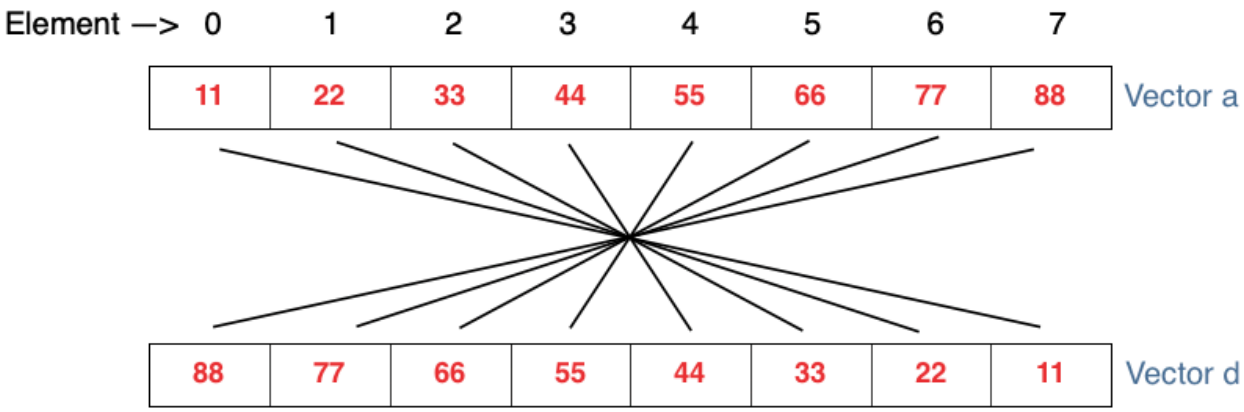
Vector Reverse

Syntax: vec_rev $vector d, $vector a

Description:

Reverses the order of the elements in vector a and stores the change in vector d

Image:



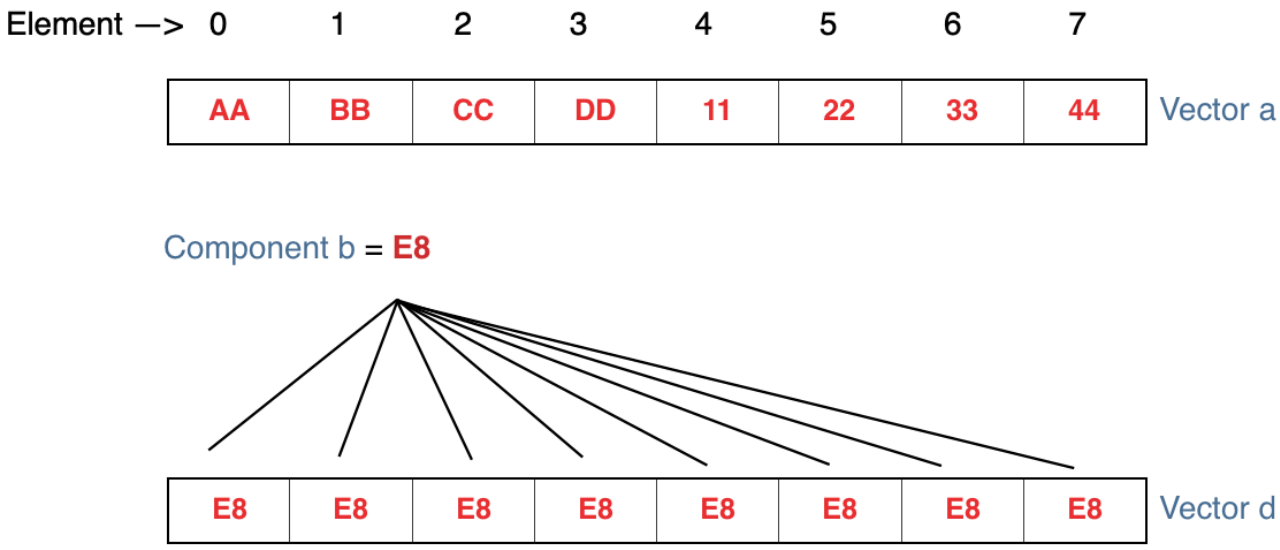
Vector Replace All

Syntax: vec_repl $vector d, $vector a, $component b

Description:

All the 8-bit elements in vector a are replaced by 8 bits stored in component b. The result is stored in vector d.

Image:



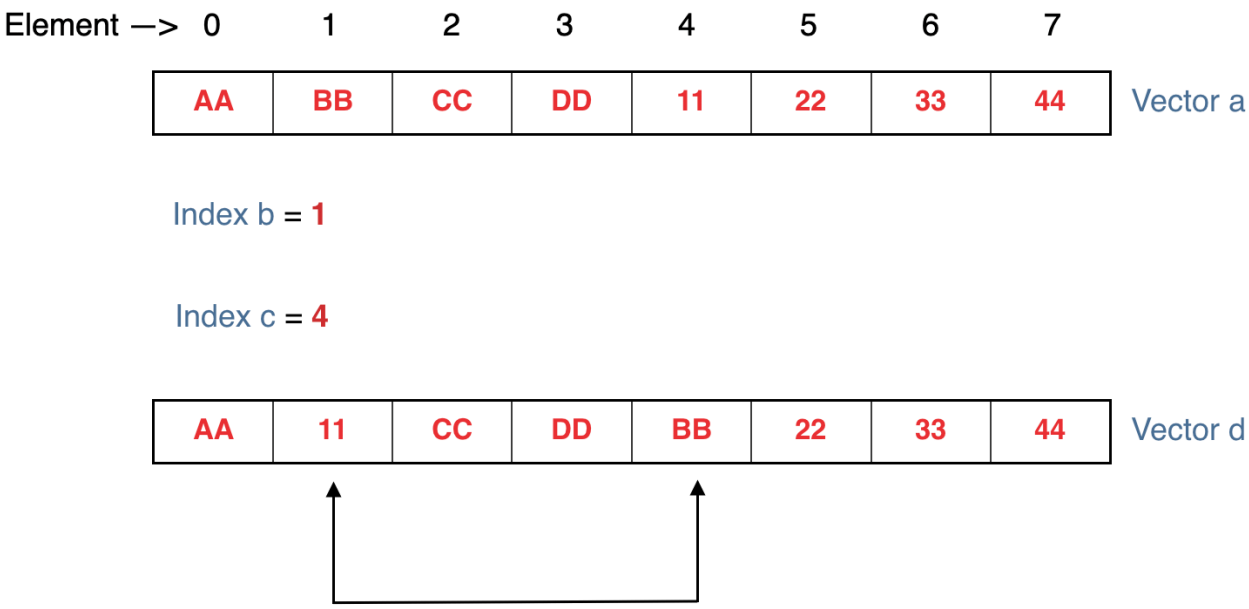
Vector Swap Element

Syntax: vec_swap_el $vector d, $vector a, $index b, $index c

Description:

Swaps the element at index b of vector a with the element at index c of vector a. The result is stored in vector d

Image:



III. MIPS Implementation/ Verification with Annotation

Vector Add Saturated (unsigned)

```
# ********************************341 Top Level Module*****************************
# File name:        vec_addsu.asm
# Version:          1.0
# Date:             December 5, 2018
# Programmer:       Thanh Le and Steven Chung
#
# Description:      Using a sequence of MIPS instructions, create a
#                   new “SIMD Enhanced" instruction that implements a Vector Add
#                   Saturated (unsigned) where the syntax is vec_addsu d, a, b. Each
#                   vector consists of eight 8-bit elements. When the instruction is
#                   executed, each element of a is added to the corresponding
#                   element of b. The unsigned-integer (no-wrap) is placed into the
#                   corresponding element of d.
#
# Register usage:   $s0, $s1, $s2, $s3, $s4, $s5, $t0, $t1, $t2, $t3
#
# Notes:            $s0 and $s1 are assumed to concatenate to indicate vector a
#                   $s2 and $s3 are assumed to concatenate to indicate vector b
#                   $s4 and $s5 are assumed to concatenate to indicate vector d
#                   $t1 and $t2 are used to store temporary data
#                   $t0 is used to take bits in an element of the vector
#                   $t3 is used as store the position (in bits) of the working element
#                   Beside $t0, $t1, $t2, $t3, $s4, and $s5, no other MIPS registers are
#                   to change
#
# ************************************************************************************
```

```
            # ******************************************************************************
            #                MAIN CODE SEGMENT
            # ******************************************************************************
            .text
            .globl  main                    # main (must be global)

main:       li      $s0, 0x233C475D         # initialize a with 4 bytes from $s0 and 4
                                              bytes from $s1
            li      $s1, 0x087F196F

            li      $s2, 0x981963C5         # initialize b with 4 bytes from $s2 and 4
                                              bytes from $s3
            li      $s3, 0X5E80B36E

            add     $s4, $zero, $zero       # clear d
            add     $s5, $zero, $zero

            addi    $t0, $zero, 0xFF        # $t0 stores the biggest 8-bit number
            add     $t3, $zero, $zero       # $t3 stores the position (in bits) of the
                                              working element

loop:       and     $t1, $s0, $t0           # load 8 bits of $s0 (an element of a)
                                              into $t1
            and     $t2, $s2, $t0           # load 8 bits in corresponding section of
                                              $s2 (an element of b) into $t2
            srlv    $t1, $t1, $t3           # shift right $t1 by the value stored in
                                              $t3 to prevent overflow
            srlv    $t2, $t2, $t3           # shift right $t2 by the value stored in
                                              $t3 to prevent overflow
            add     $t1, $t1, $t2           # add these two 8-bit values
            slti    $t2, $t1, 0xFF          # check if the result is overflow
            bne     $t2, $zero, noOverFlow1 # if NO OVERFLOW, jump to
                                              noOverFlow1
```

```
                addi   $t1, $zero, 0xFF          # if OVERFLOW, $t1 is set to FF

noOverFlow1: sllv   $t1, $t1, $t3                # shift left $t1 back to the previous
                                                   working position
                add   $s4, $s4, $t1              # store the result to the destination
                                                   register

                and   $t1, $s1, $t0              # load 8 bits of $s1 (an element of a)
                                                   into $t1
                and   $t2, $s3, $t0              # load 8 bits in corresponding section of
                                                   $s3 (an element of b) into $t2
                srlv   $t1, $t1, $t3             # shift right $t1 by the value stored in
                                                   $t3 to prevent overflow
                srlv   $t2, $t2, $t3             # shift right $t2 by the value stored in
                                                   $t3 to prevent overflow
                add   $t1, $t1, $t2              # add these two 8-bit values

                slti   $t2, $t1, 0xFF            # check if the result is overflow
                bne   $ t2, $zero, noOverFlow2   # if NO OVERFLOW, jump to
                                                   noOverFlow2
                addi  $t1, $zero, 0xFF           # if OVERFLOW, the result is set to FF

noOverFlow2: sllv    $t1, $t1, $t3          # shift left $t1 back to the previous
                                              working position
                add   $s5, $s5, $t1         # store the result to the destination register

                sll    $t0, $t0, 8          # shift $t0 left 8 bit to move to the position of
                                              the next element
                addi  $t3, $t3, 8           # increase $t3 8 bits to point to the position of
                                              the next element
                bne   $t0, $zero, loop      # if $t0 is not zero, jump to the loop
```

```
                # Clear $t0, $t1, $t2, and $t3 after finishing the execution
                add     $t0, $zero, $zero
                add     $t1, $zero, $zero
                add     $t2, $zero, $zero
                add     $t3, $zero, $zero


                # ----------------------------------------------------------------------------------------
                # "Due diligence" to return control to the kernel
                # ----------------------------------------------------------------------------------------
exit:           ori     $v0, $zero, 10              # $v0 <-- function code for "exit"
                syscall                             # Syscall to exit


                # ********************************************************************************
                #               P R O J E C T   R E L A T E D   D A T A   S E C T I O N
                # ********************************************************************************
                .data           # place variables, arrays, and constants, etc. in this area
```

Before:

Name	Number	Value
$zero	0	0x00000000
$at	1	0x5e800000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x233c475d
$s1	17	0x087f196f
$s2	18	0x981963c5
$s3	19	0x5e80b36e
$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		0x00400028
hi		0x00000000
lo		0x00000000

After:

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	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x233c475d
$s1	17	0x087f196f
$s2	18	0x981963c5
$s3	19	0x5e80b36e
$s4	20	0xbb55aaff
$s5	21	0x66ffccdd
$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		0x004000a4
hi		0x00000000
lo		0x00000000

Vector Multiply and Add

```
# ***************************341 Top Level Module*****************************
# File name:        vec_madd.asm
# Version:          1.0
# Date:             December 5, 2018
# Programmer:       Thanh Le and Steven Chung
#
# Description:      Using a sequence of MIPS instructions, create a new "SIMD
#                   Enhanced" instruction that implements a Vector Multiply and Add
#                   instruction where the syntax is vec_madd d, a, b, c. Each vector
#                   consists of eight 8-bit elements. When the instruction is executed,
#                   each elements in a is multiplied by each element in b. The
#                   intermediate result is add to the corresponding element in c, and
#                   the final sum is stored in d after being "truncated" for a half-length.
#
# Register usage:   $s0, $s1, $s2, $s3, $s4, $s5, $s6, $s7, $t0, $t1, $t2, $t3
#
# Notes:            $s0 and $s1 are assumed to concatenate to indicate vector a
#                   $s2 and $s3 are assumed to concatenate to indicate vector b
#                   $s4 and $s5 are assumed to concatenate to indicate vector c
#                   $s6 and $s7 are assumed to concatenate to indicate vector d
#                   $t1 and $t2 are used to store temporary data
#                   $t0 is used to take bits in an element of the vector
#                   $t3 is used as store the position (in bits) of the working element
#                   Beside $t0, $t1, $t2, $t3, $s6, and $s7, no other MIPS registers are
#                   to change
#
# ******************************************************************************************
```

```
# ******************************************************************************
#                MAIN CODE SEGMENT
# ******************************************************************************
.text
.globl  main                            # main (must be global)


main:           li      $s0, 0x120C1A0D     # initialize a with 4 bytes from $s0
                                              and 4 bytes from $s1
                li      $s1, 0x23051912

                li      $s2, 0x3D0C104D     # initialize b with 4 bytes from $s2
                                              and 4 bytes from $s3
                li      $s3, 0X057F192B

                li      $s4, 0x60091B05     # initialize c with 4 bytes from $s4
                                              and 4 bytes from $s5
                li      $s5, 0x501E0660

                add     $s6, $zero, $zero   # clear d
                add     $s7, $zero, $zero

                addi    $t0, $zero, 0xFF    # $t2 stores the biggest 8-bit
                                              number
                add     $t3, $zero, $zero

loop:           and     $t1, $s0, $t0       # load 8 bits of $s0 (an element
                                              of a) into $t1
                and     $t2, $s2, $t0       # load 8 bits in corresponding
                                              section of $s2 (an element of b)
                                              into $t2
```

```
                srlv    $t1, $t1, $t3           # shift right $t1 by the value
                                                  stored in $t3 to prevent
                                                  overflow
                srlv    $t2, $t2, $t3           # shift right $t2 by the value
                                                  stored in $t3 to prevent
                                                  overflow
                mult    $t1, $t2                # multiple these two 8-bit values
                                                  and store the intermediate q
                                                  result in $lo
                mflo    $t1                     # load the data in $lo into $t1
                andi    $t1, $t1, 0xFF          # truncate the result

                and     $t2, $s4, $t0           # load 8 bits in corresponding
                                                  section of $s4 (an element of c)
                                                  into $t2
                srlv    $t2, $t2, $t3           # shift right $t2 by the value
                                                  stored in $t3 to prevent
                                                  overflow
                add     $t1, $t1, $t2           # add the intermediate result
                                                  stored in $t1 to $t2
                slti    $t2, $t1, 0xFF          # check if the result is overflow
                bne     $t2, $zero, noOverFlow1 # if NO OVERFLOW, jump to
                                                  noOverFlow1
                addi    $t1, $zero, 0xFF        # if OVERFLOW, the result is set
                                                  to FF

noOverFlow1:    sllv    $t1, $t1, $t3           # shift left $t1 back to the
                                                  previous working position
                add     $s6, $s6, $t1           # store the result to the
                                                  destination register

                and     $t1, $s1, $t0           # load 8 bits of $s1 (an element
                                                  of a) in $t1
```

```
                    and     $t2, $s3, $t0           # load 8 bits in corresponding
                                                      section of $s3 (an element of b)
                                                      into $t2
                    srlv    $t1, $t1, $t3           # shift right $t1 by the value
                                                      stored in $t3 to prevent
                                                      overflow
                    srlv    $t2, $t2, $t3           # shift right $t1 by the value
                                                      stored in $t3 to prevent
                                                      overflow
                    mult    $t1, $t2                # multiple these two 8-bit and
                                                      store the intermediate result in
                                                      $lo
                    mflo    $t1                     # load the data in $lo to $t1
                    andi    $t1, $t1, 0xFF          # truncate the result

                    and     $t2, $s5, $t0           # load 8 bits in corresponding
                                                      section of $s5 (an element of c)
                                                      into $t2
                    srlv    $t2, $t2, $t3           # shift right $t2 by the value
                                                      stored in $t3 to prevent
                                                      overflow
                    add     $t1, $t1, $t2           # add the intermediate result to
                                                      $t2
                    slti    $t2, $t1, 0xFF          # check if the result is overflow
                    bne     $t2, $zero, noOverFlow2 # if NO OVERFLOW, jump to
                                                      noOverFlow2
                    add     $t1, $zero, 0xFF        # if OVERFLOW, the result is set
                                                      to FF

noOverFlow2:        sllv    $t1, $t1, $t3           # shift left $t1 back to the
                                                      previous working position
                    add     $s7, $s7, $t1           # store the result to the
                                                      destination register
```

```
        sll     $t0, $t0, 8             # shift $t0 left 8 bit to move to the
                                          position of the next element
        addi    $t3, $t3, 8             # increase $t3 8 bits to point to
                                          the position of the next element
        bne     $t0, $zero, loop        # if $t0 is not zero, jump to the
                                          loop


    # Clear $t0, $t1, $t2, and $t3 after finishing the execution
    add     $t0, $zero, $zero
    add     $t1, $zero, $zero
    add     $t2, $zero, $zero
    add     $t3, $zero, $zero


    # ---------------------------------------------------------------------------
    # "Due diligence" to return control to the kernel
    # ---------------------------------------------------------------------------
exit:   ori     $v0, $zero, 10          # $v0 <-- function code for "exit"
    syscall                             # Syscall to exit


    # ***************************************************************************
    #               P R O J E C T   R E L A T E D   D A T A   S E C T I O N
    # ***************************************************************************
    .data                       # place variables, arrays, and constants, etc. in
                                  this area
```

Before:

Name	Number	Value
$zero	0	0x00000000
$at	1	0x501e0000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x120c1a0d
$s1	17	0x23051912
$s2	18	0x3d0c104d
$s3	19	0x057f192b
$s4	20	0x60091b05
$s5	21	0x501e0660
$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		0x00400038
hi		0x00000000
lo		0x00000000

After:

Name	Number	Value
$zero	0	0x00000000
$at	1	0x501e0000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x120c1a0d
$s1	17	0x23051912
$s2	18	0x3d0c104d
$s3	19	0x057f192b
$s4	20	0x60091b05
$s5	21	0x501e0660
$s6	22	0xaa99bbee
$s7	23	0xff997766
$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		0x004000dc
hi		0x00000000
lo		0x000000af

Vector Multiply Even Integer

```
# ***************************341 Top Level Module*****************************
# File name:        vec_mule.asm
# Version:          1.0
# Date:             December 5, 2018
# Programmer:       Thanh Le and Steven Chung
#
# Description:      Using a sequence of MIPS instructions, create a new "SIMD
#                   Enhanced" instruction that implements a Vector Multiply Even
#                   Integer where the syntax is vec_mule d, a, b. Vector a and b consist
#                   of eight 8-bit elements. Vector d consists of four 16-bit elements.
#                   When the instruction is executed, each even element of a is added
#                   to the corresponding even element of b. The result is stored in full-
#                   length (16-bit) in each element of vector d.
#
# Register usage:   $s0, $s1, $s2, $s3, $s4, $s5, $t0, $t1, $t2, $t3
#
# Notes:            $s0 and $s1 are assumed to concatenate to indicate vector a
#                   $s2 and $s3 are assumed to concatenate to indicate vector b
#                   $s4 and $s5 are assumed to concatenate to indicate vector d
#                   $t1 and $t2 are used to store temporary data
#                   $t0 is used to take bits in an element of the vector
#                   $t3 is used as store the position (in bits) of the working element
#                   Beside $t0, $t1, $t2, $t3, $s4, and $s5, no other MIPS registers are
#                   to change
#
# ******************************************************************************************
```

```
# ******************************************************************************
#                MAIN CODE SEGMENT
# ******************************************************************************
.text
.globl  main                    # main (must be global)


main:   li      $s0, 0xAEE95AE0 # initialize a with 4 bytes from $s0 and 4 bytes
                                  from $s1
        li      $s1, 0xF080CC66

        li      $s2, 0x33146170 # initialize b with 4 bytes from $s2 and 4 bytes
                                  from $s3
        li      $s3, 0X609888AB

        add     $s4, $zero, $zero # clear d
        add     $s5, $zero, $zero

        addi    $t0, $zero, 0xFF00 # $t0 stores the biggest 4-bit number
        add     $t3, $zero, $zero # $t3 stores the position (in bits) of the working
                                  element


loop:   and     $t1, $s0, $t0   # load 8 bits of $s0 (an element of a) into $t1
        and     $t2, $s2, $t0   # load 8 bits in corresponding section of $s2
                                  (an element of b) into $t2
        srlv    $t1, $t1, $t3   # shift right $t1 by the value stored in $t3 to
                                  prevent overflow
        srlv    $t2, $t2, $t3   # shift right $t2 by the value stored in $t3 to
                                  prevent overflow
        mult    $t1, $t2        # multiply these two 8-bit values
        mflo    $t1             # load the data in $lo into $t1
```

```
addi    $t3, $t3, 16        # adjust the pointer $t3
srlv    $t1, $t1, $t3       # shift right the result to the proper position of
                              the element
addi    $t3, $t3, -16       # move the pointer $t3 back to the previous
                              position
add     $s4, $s4, $t1       # store the result to the destination register

and     $t1, $s1, $t0       # load 8 bits of $s1 (an element of a) into $t1
and     $t2, $s3, $t0       # load 8 bits in corresponding section of $s3
                              (an element of b) into $t2
srlv    $t1, $t1, $t3       # shift right $t1 by the value stored in $t3 to
                              prevent overflow
srlv    $t2, $t2, $t3       # shift right $t2 by the value stored in $t3 to
                              prevent overflow
mult    $t1, $t2            # multiply these two 8-bit values and store the
                              value in $lo
mflo    $t1                 # load the data in $lo into $t1
addi    $t3, $t3, 16        # adjust the pointer $t3
srlv    $t1, $t1, $t3       # shift right the result to the proper position of
                              the element
addi    $t3, $t3, -16       # move the pointer $t3 back to the previous
                              position
add     $s5, $s5, $t1       # store the result to the destination register

sll     $t0, $t0, 16        # shift $t0 left 8 bit to move to the position of
                              the next element
addi    $t3, $t3, 16        # increase $t3 8 bits to point to the position of
                              the next element
bne     $t0, $zero, loop    # if $t0 is not zero, jump to the loop
```

```
                # Clear $t0, $t1, $t2, and $t3 after finishing the execution
                add     $t0, $zero, $zero
                add     $t1, $zero, $zero
                add     $t2, $zero, $zero
                add     $t3, $zero, $zero



                # -----------------------------------------------------------------------------
                # "Due diligence" to return control to the kernel
                # -----------------------------------------------------------------------------
exit:           ori     $v0, $zero, 10      # $v0 <-- function code for "exit"
                syscall                     # Syscall to exit



                # *****************************************************************************
                #               P R O J E C T   R E L A T E D   D A T A   S E C T I O N
                # *****************************************************************************
.data                                       # place variables, arrays, and constants, etc. in
                                              this area
```

Before:

Name	Number	Value
$zero	0	0x00000000
$at	1	0x60980000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0xaee95ae0
$s1	17	0xf080cc66
$s2	18	0x33146170
$s3	19	0x609888ab
$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		0x00400028
hi		0x00000000
lo		0x00000000

After:

Name	Number	Value
$zero	0	0x00000000
$at	1	0x0000ff00
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0xaee95ae0
$s1	17	0xf080cc66
$s2	18	0x33146170
$s3	19	0x609888ab
$s4	20	0x22aa221a
$s5	21	0x5a006c60
$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		0x004000ac
hi		0x00000000
lo		0x5a000000

Vector Multiply Odd Integer

```
# ***************************341 Top Level Module*****************************
# File name:        vec_mulo.asm
# Version:          1.0
# Date:             December 5, 2018
# Programmer:       Thanh Le and Steven Chung
#
# Description:      Using a sequence of MIPS instructions, create a new "SIMD
#                   Enhanced" instruction that implements a Vector Multiply Odd
#                   Integer where the syntax is vec_mulo d, a, b. Vector a and b consist
#                   of eight 8-bit elements. Vector d consists of four 16-bit elements.
#                   When the instruction is executed, each even element of a is added
#                   to the corresponding even element of b. The result is stored in full-
#                   length (16-bit) in each element of vector d.
#
# Register usage:   $s0, $s1, $s2, $s3, $s4, $s5, $t0, $t1, $t2, $t3
#
# Notes:            $s0 and $s1 are assumed to concatenate to indicate vector a
#                   $s2 and $s3 are assumed to concatenate to indicate vector b
#                   $s4 and $s5 are assumed to concatenate to indicate vector d
#                   $t1 and $t2 are used to store temporary data
#                   $t0 is used to take bits in an element of the vector
#                   $t3 is used as store the position (in bits) of the working element
#                   Beside $t0, $t1, $t2, $t3, $s4, and $s5, no other MIPS registers are
#                   to change
#
# ******************************************************************************************
```

```
# ******************************************************************************
#                MAIN CODE SEGMENT
# ******************************************************************************
.text
.globl  main                    # main (must be global)


main:   li      $s0, 0xAEE95AE0 # initialize a with 4 bytes from $s0 and 4 bytes
                                  from $s1
        li      $s1, 0xF080CC66

        li      $s2, 0x33146170 # initialize b with 4 bytes from $s2 and 4 bytes
                                  from $s3
        li      $s3, 0X609888AB

        add     $s4, $zero, $zero # clear d
        add     $s5, $zero, $zero

        addi    $t0, $zero, 0xFF  # $t0 stores the biggest 4-bit number
        add     $t3, $zero, $zero # $t3 stores the position (in bits) of the working
                                  element


loop:   and     $t1, $s0, $t0     # load 8 bits of $s0 (an element of a) into $t1
        and     $t2, $s2, $t0     # load 8 bits in corresponding section of $s2
                                  (an element of b) into $t2
        srlv    $t1, $t1, $t3     # shift right $t1 by the value stored in $t3 to
                                  prevent overflow
        srlv    $t2, $t2, $t3     # shift right $t2 by the value stored in $t3 to
                                  prevent overflow
        mult    $t1, $t2          # multiply these two 8-bit values
        mflo    $t1               # load the data in $lo into $t1
```

```
sllv    $t1, $t1, $t3       # shift left $t1 back to the previous working
                              position
add     $s4, $s4, $t1       # store the result to the destination register

and     $t1, $s1, $t0       # load 8 bits of $s1 (an element of a) into $t1
and     $t2, $s3, $t0       # load 8 bits in corresponding section of $s3
                              (an element of b) into $t2
srlv    $t1, $t1, $t3       # shift right $t1 by the value stored in $t3 to
                              prevent overflow
srlv    $t2, $t2, $t3       # shift right $t2 by the value stored in $t3 to
                              prevent overflow
mult    $t1, $t2            # multiply these two 8-bit values and store the
                              value in $lo
mflo    $t1                 # load the data in $lo into $t1
sllv    $t1, $t1, $t3       # shift left $t1 back to the previous working
                              position
add     $s5, $s5, $t1       # store the result to the destination register

sll     $t0, $t0, 16        # shift $t0 left 8 bit to move to the position of
                              the next element
addi    $t3, $t3, 16        # increase $t3 8 bits to point to the position of
                              the next element
bne     $t0, $zero, loop    # if $t0 is not zero, jump to the loop


# Clear $t0, $t1, $t2, and $t3 after finishing the execution
add     $t0, $zero, $zero
add     $t1, $zero, $zero
add     $t2, $zero, $zero
add     $t3, $zero, $zero
```

```
                # ---------------------------------------------------------------------------
                # "Due diligence" to return control to the kernel
                # ---------------------------------------------------------------------------
exit:           ori     $v0, $zero, 10      # $v0 <-- function code for "exit"
                syscall                     # Syscall to exit


                # ***************************************************************************
                #               P R O J E C T   R E L A T E D   D A T A   S E C T I O N
                # ***************************************************************************
                .data                       # place variables, arrays, and constants, etc. in
                                              this area
```

Before:

Name	Number	Value
$zero	0	0x00000000
$at	1	0x60980000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0xaee95ae0
$s1	17	0xf080cc66
$s2	18	0x33146170
$s3	19	0x609888ab
$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		0x00400028
hi		0x00000000
lo		0x00000000

After:

Name	Number	Value
$zero	0	0x00000000
$at	1	0x60980000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0xaee95ae0
$s1	17	0xf080cc66
$s2	18	0x33146170
$s3	19	0x609888ab
$s4	20	0x12346200
$s5	21	0x4c004422
$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		0x00400094
hi		0x00000000
lo		0x00004c00

Vector Multiply Sum Saturated

```
# ***************************341 Top Level Module*****************************
# File name:        vec_msums.asm
# Version:          1.0
# Date:             December 5, 2018
# Programmer:       Thanh Le and Steven Chung
#
# Description:      Using a sequence of MIPS instructions, create a new "SIMD
#                   Enhanced" instruction that implements a Vector Multiply Sum
#                   Saturated instruction where the syntax is vec_msums d, a, b, c.
#                   Vector a and b consist of eight 8-bit elements. Vector c and d
#                   consist of four 16-bit elements. When the instruction is executed,
#                   each element of vector d is the 16-bit sum of the corresponding
#                   elements of vector and the 16-bit “temp" products of the 8-bit
#                   elements of vector a and vector b which overlap the positions of
#                   that element in c. The sum is performed with 16-bit saturating
#                   addition (no-wrap)
#
# Register usage:   $s0, $s1, $s2, $s3, $s4, $s5, $s6, $s7, $t0, $t1, $t2, $t3, $t4, $v0,
#                   $v1
#
# Notes:            $s0 and $s1 are assumed to concatenate to indicate vector a
#                   $s2 and $s3 are assumed to concatenate to indicate vector b
#                   $s4 and $s5 are assumed to concatenate to indicate vector c
#                   $s6 and $s7 are assumed to concatenate to indicate vector d
#                   $t1 and $t2 are used to store temporary data
#                   $t0, $t3, $t4, and $t5 are used to point to a specific elements or bits
#                   $v0 and $v1 are used to store working result
#
#
# ******************************************************************************************
```

```
# ******************************************************************************
#                MAIN CODE SEGMENT
# ******************************************************************************
.text
.globl  main                    # main (must be global)


main:   li      $s0, 0x230CF14D     # initialize a with 4 bytes from $s0 and 4
                                      bytes from $s1
        li      $s1, 0x5C7F191A

        li      $s2, 0xA30C5BFD     # initialize b with 4 bytes from $s2 and 4
                                      bytes from $s3
        li      $s3, 0XC5FFC9EE

        li      $s4, 0x609E19F7     # initialize c with 4 bytes from $s4 and 4
                                      bytes from $s5
        li      $s5, 0x45670766

        add   $s6, $zero, $zero     # clear d
        add   $s7, $zero, $zero

        addi  $t0, $zero, 0xFF      # $t2 stores the biggest 8-bit number
        add   $t3, $zero, $zero
        addi  $t4, $zero, 0xFFFF
        add   $t5, $zero, $zero

loop:   # For 32 lower bits, multiply odd elements of vector a and vector b
        and   $t1, $s0, $t0         # load 8 bits of $s0 (an element of a)
                                      into $t1
        and   $t2, $s2, $t0         # load 8 bits in corresponding section of
                                      $s2 (an element of b) into $t2
```

```
srlv   $t1, $t1, $t3          # shift right $t1 by the value stored in
                                $t3 to prevent overflow
srlv   $t2, $t2, $t3          # shift right $t2 by the value stored in
                                $t3 to prevent overflow
mult   $t1, $t2               # multiply these two 8-bit values
mflo   $t1                    # load the data in $lo into $t1
sllv   $t1, $t1, $t3          # shift left $t1 back to the previous
                                working position
add    $v0, $v0, $t1          # store the result to the destination
                                register

# For 32 upper bits, repeat the logic to calculate the odd multiplication of
  upper 32 bits of vector a and vector b
and    $t1, $s1, $t0
and    $t2, $s3, $t0
srlv   $t1, $t1, $t3
srlv   $t2, $t2, $t3
mult   $t1, $t2
mflo   $t1
sllv   $t1, $t1, $t3
add    $v1, $v1, $t1

# Move the $t0 to the position of even position
sll    $t0, $t0, 8

# For 32 lower bits, multiply even elements of vector a and b and add the
  result to the corresponding element of vector c
and    $t1, $s0, $t0          # load 8 bits of $s0 (an element of a)
                                into $t1
and    $t2, $s2, $t0          # load 8 bits in corresponding section of
                                $s2 (an element of b) into $t2
srlv   $t1, $t1, $t3          # shift right $t1 by the value stored in
                                $t3 to prevent overflow
```

```
                srlv   $t2, $t2, $t3        # shift right $t2 by the value stored in
                                              $t3 to prevent overflow
                mult   $t1, $t2             # multiply these two 8-bit values
                mflo   $t1                  # load the data in $lo into $t1
                addi   $t3, $t3, 16         # adjust the pointer $t3
                srlv   $t1, $t1, $t3        # shift right the result to the proper
                                              position of the element
                addi   $t3, $t3, -16        # move the pointer $t3 back to the
                                              previous position

                and    $t2, $v0, $t4        # load the result of the odd multiplication
                                              into $t2
                srlv   $t1, $t1, $t5        # shift right $t1 by the value stored in
                                              $t5 to prevent overflow
                srlv   $t2, $t2, $t5        # shift right $t2 by the value stored in
                                              $t5 to prevent overflow
                add    $t1, $t1, $t2        # add the even multiplication and odd
                                              multiplication

                addi   $t2, $zero, 0xFFFF   # store the biggest 16-bit value in $t2
                slt    $t2, $t1, $t2        # check if the result is overflow
                bne    $t2, $zero, noOverFlow1 # if NO OVERFLOW, jump to
                                              noOverFlow1
                addi   $t1, $zero, 0xFFFF   # if OVERFLOW, the result is set to
                                              FFFF

noOverFlow1:    and    $t2, $s4, $t4        # load 16 bits in corresponding section
                                              of $s2 (an element of c) into $t2
                srlv   $t2, $t2, $t5        # shift right $t2 by the value stored in
                                              $t3 to prevent overflow
                add    $t1, $t1, $t2        # add the 16-bit sum of the odd and
                                              even multiplications of a and b to the
                                              16-bit element of c
```

```
                addi  $t2, $zero, 0xFFFF        # store the biggest 16-bit value in $t2
                slt   $t2, $t1, $t2             # check if the result is overflow
                bne   $t2, $zero, noOverFlow2   # if NO OVERFLOW, jump to
                                                  noOverFlow2
                addi  $t1, $zero, 0xFFFF        # if OVERFLOW, the result is set to
                                                  FFFF

noOverFlow2: sllv   $t1, $t1, $t5               # shift left $t1 back to the previous
                                                  working position
                add   $s6, $s6, $t1             # store the result to the proper element
                                                  of the destination vector d

               # For 32 upper bits, repeat the logic to calculate the even multiplication,
                 add it to the odd multiplication, and add the result to the corresponding
                 element of vector c
                and   $t1, $s1, $t0
                and   $t2, $s3, $t0
                srlv  $t1, $t1, $t3
                srlv  $t2, $t2, $t3
                mult  $t1, $t2
                mflo  $t1
                addi  $t3, $t3, 16
                srlv  $t1, $t1, $t3
                addi  $t3, $t3, -16

                and   $t2, $v1, $t4
                srlv  $t1, $t1, $t5
                srlv  $t2, $t2, $t5
                add   $t1, $t1, $t2

                addi  $t2, $zero, 0xFFFF
```

```
                slt     $t2, $t1, $t2
                bne     $t2, $zero, noOverFlow3
                addi    $t1, $zero, 0xFFFF

noOverFlow3: and    $t2, $s5, $t4
                srlv    $t2, $t2, $t5
                add     $t1, $t1, $t2

                addi    $t2, $zero, 0xFFFF
                slt     $t2, $t1, $t2
                bne     $t2, $zero, noOverFlow4
                addi    $t1, $zero, 0xFFFF

noOverFlow4: sllv   $t1, $t1, $t5
                add     $s7, $s7, $t1


                # Adjust the bits value of $t0, $t3, $4, and $t5 to point to the next working
                element
                  sll     $t0, $t0, 8
                  addi    $t3, $t3, 16
                  sll     $t4, $t4, 16
                  addi    $t5, $t5, 16
                  bne     $t0, $zero, loop      # loop until the $t0 is shift all to the left, and its
                                                  value become 0


                # Clear $t0, $t1, $t2, and $t3 after finishing the execution
                  add     $t0, $zero, $zero
                  add     $t1, $zero, $zero
                  add     $t2, $zero, $zero
```

```
        add    $t3, $zero, $zero
        add    $v0, $zero, $zero
        add    $v1, $zero, $zero


        # ------------------------------------------------------------------------------
        # "Due diligence" to return control to the kernel
        # ------------------------------------------------------------------------------
exit:   ori    $v0, $zero, 10      # $v0 <-- function code for "exit"
        syscall                    # Syscall to exit


        # ******************************************************************************
        #              P R O J E C T   R E L A T E D   D A T A   S E C T I O N
        # ******************************************************************************
        .data                      # place variables, arrays, and constants, etc. in
                                     this area
```

Before:

Name	Number	Value
$zero	0	0x00000000
$at	1	0x45670000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x230cf14d
$s1	17	0x5c7f191a
$s2	18	0xa30c5bfd
$s3	19	0xc5ffc9ee
$s4	20	0x609e19f7
$s5	21	0x45670766
$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		0x00400038
hi		0x00000000
lo		0x00000000

After:

Name	Number	Value
$zero	0	0x00000000
$at	1	0x0000ffff
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000020
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x230cf14d
$s1	17	0x5c7f191a
$s2	18	0xa30c5bfd
$s3	19	0xc5ffc9ee
$s4	20	0x609e19f7
$s5	21	0x45670766
$s6	22	0x7777bbbb
$s7	23	0xffff3333
$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		0x004001d8
hi		0x00000000
lo		0x46cc0000

Vector Splat

```
# ***************************341 Top Level Module*****************************
# File name:        vec_splat.asm
# Version:          1.0
# Date:             December 5, 2018
# Programmer:       Thanh Le and Steven Chung
#
# Description:      Using a sequence of MIPS instructions, create a new "SIMD
#                   Enhanced" instruction that implements a Vector Splat instruction
#                   where the syntax is vec_splat d, a, b. Vector a and d consist of
#                   eight 8-bit elements. When the instruction is execute, it copies any
#                   element which is indicated by component b from vector a into all of
#                   the elements of vector d.
#
# Register usage:   $s0, $s1, $s2, $s3, $s4, $t0, $t1, $t2, $t3, $t4
#
# Notes:            $s0 and $s1 are assumed to concatenate to indicate vector a
#                   $s2 is the index of the element b to be copied
#                   $s4 and $s5 are assumed to concatenate to indicate vector d
#                   $t0, $t1, $t2, $t3, and $t4 are used to store temporary data
#
# ******************************************************************************************
```

```
            # ******************************************************************************
            #                MAIN CODE SEGMENT
            # ******************************************************************************
            .text
            .globl  main                    # main (must be global)


main:       li      $s0, 0x230C124D         # initialize a with 4 bytes from $s0 and 4
                                              bytes from $s1
            li      $s1, 0x057F192A

            li      $s2, 5                  # initialize the index of the element to be
copied

            add     $s3, $zero, $zero       # clear d
            add     $s4, $zero, $zero

            addi    $t0, $zero, 0xFF000000
            addi    $t1, $zero, 0x8         # $t1 is assumed to be a pointer to get
                                              8-bit data at an index
            addi    $t2, $zero, 0x18        # $t2 stores the number of bits to be
                                              shift to reach the last element of 32-bit
                                              vector segment
            addi    $t3, $zero, 0x4         # $t3 stores the number of loops needed
                                              for this instruction
                                            # (loop 4 times since there are 4
                                              element in each 32-bit vector
                                              segment)
            add     $t4, $zero, $s2
            bgt     $t4, 3, upper           # if the index is greater than 3, jump to
                                              upper
            j       lower                   # if the index is not greater than 3, jump
                                              to lower
```

```
                # Determine the data to be copied at the index b
                # When the index b is greater than 3
upper:          addi    $t4, $t4, -4            # assume the index to be counted from
                                                  the first element of the upper vector
                                                  segment
                mult    $t1, $t4                # calculate the number of bits to be
                                                  shifted to move the pointer $t0 to the
                                                  index
                mflo    $t1
                srlv    $t0, $t0, $t1           # shift the pointer $t0 to the index
                and     $t0, $s1, $t0           # get the data at the index
                j       shift

                # When the index b is smaller or equal to 3
lower:          mult    $t1, $t4                # calculate the number of bits to be
                                                  shifted to move the pointer $t0 to the
                                                  index
                mflo    $t1
                srlv    $t0, $t0, $t1           # shift the pointer $t0 to the index
                and     $t0, $s0, $t0           # get the data at the index

                # Shift the data to be copied to the last element of the vector
shift:          sub     $t2, $t2, $t1           # calculate the number of bits to be
                                                  shifted to reach the last element of
                                                  the 32-bit vector segment
                srlv    $t0, $t0, $t2           # shift the data to be copied to each
                                                  element to the position of the last
                                                  element
```

```
                # Loop through every element of the vector and copy the data
loop:           add     $s3, $s3, $t0           # copy the data to an element of the
                                                  lower 32-bit destination vector
                                                  segment
                add     $s4, $s4, $t0           # copy the data to an element of the
                                                  upper 32-bit destination vector
                                                  segment
                sll     $t0, $t0, 8             # move the pointer the next element
                addi    $t3, $t3, -1            # decrease the number of loops needed
                                                  by 1
                bne     $t3, $zero, loop        # loop 4 times since there are 4 element
                                                  in each 32-bit vector segment
                                                # loop until the counter $t3 is equal to 0

                # Clear $t0, $t1, $t2, and $t3 after finishing the execution
                add     $t0, $zero, $zero
                add     $t1, $zero, $zero
                add     $t2, $zero, $zero
                add     $t3, $zero, $zero
                add     $t4, $zero, $zero

                # ---------------------------------------------------------------------------
                # "Due diligence" to return control to the kernel
                # ---------------------------------------------------------------------------
exit:           ori     $v0, $zero, 10          # $v0 <-- function code for "exit"
                syscall                         # Syscall to exit

                # ***************************************************************************
                #               P R O J E C T   R E L A T E D   D A T A   S E C T I O N
                # ***************************************************************************
                .data                       # place variables, arrays, and constants, etc. in
                                              this area
```

Before:

Name	Number	Value
$zero	0	0x00000000
$at	1	0x057f0000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x230c124d
$s1	17	0x057f192a
$s2	18	0x00000005
$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		0x0040001c
hi		0x00000000
lo		0x00000000

After:

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000001
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x230c124d
$s1	17	0x057f192a
$s2	18	0x00000005
$s3	19	0x7f7f7f7f
$s4	20	0x7f7f7f7f
$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		0x004000a8
hi		0x00000000
lo		0x00000008

Vector Merge Low

```
# ***************************341 Top Level Module*****************************
# File name:        vec_mergel.asm
# Version:          1.0
# Date:             December 5, 2018
# Programmer:       Thanh Le and Steven Chung
#
# Description:      Using a sequence of MIPS instructions, create a new "SIMD
#                   Enhanced" instruction that implements a Vector Merge Low
#                   instruction where the syntax is vec_mergel d, a, b. Vector a, b, and
#                   d consist of eight 8-bit elements. When the instruction is execute,
#                   the even elements of the result vector d are obtained left-to-right
#                   from the low elements of vector a. The odd elements of the result
#                   are obtained left-to-right from the low elements of vector b.
#
# Register usage:   $s0, $s1, $s2, $s3, $s4, $s5, $t0, $t1, $t2, $t3
#
# Notes:            $s0 and $s1 are assumed to concatenate to indicate vector a
#                   $s2 and $s3 are assumed to concatenate to indicate vector b
#                   $s4 and $s5 are assumed to concatenate to indicate vector d
#                   $t0, $t1, $t2, and $t3 are used to store temporary data
#
# ******************************************************************************************
```

```
# ******************************************************************************
#                MAIN CODE SEGMENT
# ******************************************************************************
.text
.globl  main                    # main (must be global)


main:   li      $s0, 0x5AF0A501         # initialize a with 4 bytes from $s0 and 4
                                          bytes from $s1
        li      $s1, 0xAB0155C3

        li      $s2, 0xA50F5A23         # initialize b with 4 bytes from $s2 and 4
                                          bytes from $s3
        li      $s3, 0xCD23AA3C

        add     $s4, $zero, $zero       # clear d
        add     $s5, $zero, $zero

        addi    $t0, $zero, 0xFF
        addi    $t3, $zero, 0x4         # $t3 stores the number of loop needed
                                          for this instruction
                                        # (loop 4 times since there are 4
                                          element in each 32-bit vector
                                          segment)
        add     $t4, $zero, $zero       # $t4 acts as pointer to shift data


loop:   and     $t1, $s1, $t0           # load 8 bits of $s1 (a lower element of
                                          a) into $t1
        and     $t2, $s3, $t0           # load 8 bits in corresponding section of
                                          $s3 (a lower element of b) into $t
```

```
            srlv    $t1, $t1, $t4       # shift right $t1 to the last element of the
                                          vector
            srlv    $t2, $t2, $t4       # shift right $t2 to the last element of the
                                          vector
            sll     $t1, $t1, 8         # shift $t1 left 8 bits to move the data to
                                          the next last element
            add     $t1, $t1, $t2       # add these data to form a section of 16-
                                          bit where the element of vector a
                                        # stay in the even index while the
                                          element of vector b stay in the odd
                                          index

            sll     $t4, $t4, 1         # adjust the value of pointer $t4 to shift
                                          the 16-bit result to proper position
            sllv    $t1, $t1, $t4
            srl     $t4, $t4, 1

            bgt     $t3, 2, upper       # if the index is greater than 2, consider
                                          the upper 32-bit vector segment of
                                          vector d

lower:      add     $s4, $s4, $t1       # add the result to the element of the
                                          lower 32-bit vector segment of vector
                                          d
            j       jump

upper:      add     $s5, $s5, $t1       # add the result to the element of the
                                          upper 32-bit vector segment of vector
                                          d

jump:       addi    $t3, $t3, -1        # decrease the number of loop left to
                                          finish this instruction
            sll     $t0, $t0, 8         # adjust the pointer $t0 to the next
                                          element
```

```
        addi    $t4, $t4, 8             # increase the number of bits needed to
                                          shift the next element to the last
                                          position in the vector
        bne     $t3, $zero, loop        # loop until the counter $t3 is equal to 0


        # Clear $t0, $t1, $t2, and $t3 after finishing the execution
        add     $t0, $zero, $zero
        add     $t1, $zero, $zero
        add     $t2, $zero, $zero
        add     $t3, $zero, $zero


        # ---------------------------------------------------------------------------
        # "Due diligence" to return control to the kernel
        # ---------------------------------------------------------------------------
exit:   ori     $v0, $zero, 10          # $v0 <-- function code for "exit"
        syscall                         # Syscall to exit


        # ***************************************************************************
        #               P R O J E C T   R E L A T E D   D A T A   S E C T I O N
        # ***************************************************************************
        .data                           # place variables, arrays, and constants,
                                          etc. in this area
```

Before:

Name	Number	Value
$zero	0	0x00000000
$at	1	0xcd230000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x5af0a501
$s1	17	0xab0155c3
$s2	18	0xa50f5a23
$s3	19	0xcd23aa3c
$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		0x00400028
hi		0x00000000
lo		0x00000000

After:

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000020
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x5af0a501
$s1	17	0xab0155c3
$s2	18	0xa50f5a23
$s3	19	0xcd23aa3c
$s4	20	0xabcd0123
$s5	21	0x55aac33c
$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		0x00400098
hi		0x00000000
lo		0x00000000

Vector Merge High

```
# ***************************341 Top Level Module*****************************
# File name:        vec_mergeh.asm
# Version:          1.0
# Date:             December 5, 2018
# Programmer:       Thanh Le and Steven Chung
#
# Description:      Using a sequence of MIPS instructions, create a new "SIMD
#                   Enhanced" instruction that implements a Vector Merge High
#                   instruction where the syntax is vec_merge d, a, b. Vector a, b, and
#                   d consist of eight 8-bit elements. When the instruction is execute,
#                   the even elements of the result vector d are obtained left-to-right
#                   from the high elements of vector a. The odd elements of the result
#                   are obtained left-to-right from the high elements of vector b.
#
# Register usage:   $s0, $s1, $s2, $s3, $s4, $s5, $t0, $t1, $t2, $t3
#
# Notes:            $s0 and $s1 are assumed to concatenate to indicate vector a
#                   $s2 and $s3 are assumed to concatenate to indicate vector b
#                   $s4 and $s5 are assumed to concatenate to indicate vector d
#                   $t0, $t1, $t2, and $t3 are used to store temporary data
#
# ******************************************************************************************
```

```
                # ******************************************************************************
                #                MAIN CODE SEGMENT
                # ******************************************************************************
                .text
                .globl  main                    # main (must be global)


main:           li      $s0, 0x5AF0A501         # initialize a with 4 bytes from $s0 and 4
                                                  bytes from $s1
                li      $s1, 0xAB0155C3

                li      $s2, 0xA50F5A23         # initialize b with 4 bytes from $s2 and 4
                                                  bytes from $s3
                li      $s3, 0xCD23AA3C

                add     $s4, $zero, $zero       # clear d
                add     $s5, $zero, $zero

                addi    $t0, $zero, 0xFF
                addi    $t3, $zero, 0x4         # $t3 stores the number of loop needed
for this instruction
                                                # (loop 4 times since there are 4
                                                  element in each 32-bit vector
                                                  segment)
                add     $t4, $zero, $zero       # $t4 acts as pointer to shift data


loop:           and     $t1, $s0, $t0           # load 8 bits of $s0 (a higher element of
                                                  a) into $t1
                and     $t2, $s2, $t0           # load 8 bits in corresponding section of
                                                  $s2 (a higher element of b) into $t
                srlv    $t1, $t1, $t4           # shift right $t1 to the last element of the
                                                  vector
```

```
                srlv    $t2, $t2, $t4       # shift right $t2 to the last element of the
                                              vector
                sll     $t1, $t1, 8         # shift $t1 left 8 bits to move the data to
                                              the next last element
                add     $t1, $t1, $t2       # add these data to form a section of 16-
                                              bit where the element of vector a
                                            # stay in the even index while the
                                              element of vector b stay in the odd
index

                sll     $t4, $t4, 1         # adjust the value of pointer $t4 to shift
                                              the 16-bit result to proper position
                sllv    $t1, $t1, $t4
                srl     $t4, $t4, 1

                bgt     $t3, 2, upper       # if the index is greater than 2, conisder
                                              the upper 32-bit vector segment of
                                              vector d

lower:          add     $s4, $s4, $t1       # add the result to the element of the
                                              lower 32-bit vector segment of vector
                                              d
                j       jump

upper:          add     $s5, $s5, $t1       # add the result to the element of the
                                              upper 32-bit vector segment of vector
                                              d

jump:           addi    $t3, $t3, -1        # decrease the number of loop left to
                                              finish this instruction
                sll     $t0, $t0, 8         # adjust the pointer $t0 to the next
                                              element
```

```
        addi    $t4, $t4, 8             # increase the number of bits needed to
                                          shift the next element to the last
                                          position in the vector
        bne     $t3, $zero, loop        # loop until the counter $t3 is equal to 0


        # Clear $t0, $t1, $t2, and $t3 after finishing the execution
        add     $t0, $zero, $zero
        add     $t1, $zero, $zero
        add     $t2, $zero, $zero
        add     $t3, $zero, $zero


        # ---------------------------------------------------------------------------
        # "Due diligence" to return control to the kernel
        # ---------------------------------------------------------------------------
exit:   ori     $v0, $zero, 10          # $v0 <-- function code for "exit"
        syscall                         # Syscall to exit


        # ***************************************************************************
        #               P R O J E C T   R E L A T E D   D A T A   S E C T I O N
        # ***************************************************************************
        .data                           # place variables, arrays, and constants, etc. in
                                          this area
```

Before:

Name	Number	Value
$zero	0	0x00000000
$at	1	0xcd230000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x5af0a501
$s1	17	0xab0155c3
$s2	18	0xa50f5a23
$s3	19	0xcd23aa3c
$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		0x00400028
hi		0x00000000
lo		0x00000000

After:

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000020
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x5af0a501
$s1	17	0xab0155c3
$s2	18	0xa50f5a23
$s3	19	0xcd23aa3c
$s4	20	0x5aa5f00f
$s5	21	0xa55a0123
$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		0x00400098
hi		0x00000000
lo		0x00000000

Vector Pack

```
# ***************************341 Top Level Module*****************************
# File name:        vec_pack.asm
# Version:          1.0
# Date:             December 5, 2018
# Programmer:       Thanh Le and Steven Chung
#
# Description:      Using a sequence of MIPS instructions, create a new "SIMD
#                   Enhanced" instruction that implements a Vector Pack instruction
#                   where the syntax is vec_pack d, a, b. Vector a and b consist of
#                   eight 8-bit elements. Vector d consists of sixteen 4-bit elements.
#                   When the instruction is executed, each high element of the vector d
#                   is the truncation of the corresponding wider element of a, and each
#                   low element of the vector d is the truncation of the corresponding
#                   wider element of b.
#
# Register usage:   $s0, $s1, $s2, $s3, $s4, $s5, $t0, $t1, $t2, $t3
#
# Notes:            $s0 and $s1 are assumed to concatenate to indicate vector a
#                   $s2 and $s3 are assumed to concatenate to indicate vector b
#                   $s4 and $s5 are assumed to concatenate to indicate vector d
#                   $t1 and $t2 are used to store temporary data
#                   $t0 and $t3 are used to point to a specific elements or store value
#                   to shift
#
# ******************************************************************************************
```

```
# ******************************************************************************
#                MAIN CODE SEGMENT
# ******************************************************************************
.text
.globl  main                    # main (must be global)


main:   li      $s0, 0x5AFB6C1D  # initialize a with 4 bytes from $s0 and 4 bytes
                                   from $s1
        li      $s1, 0xAE5FC041

        li      $s2, 0x52F3A415  # initialize b with 4 bytes from $s2 and 4 bytes
                                   from $s3
        li      $s3, 0xA657C849

        add     $s4, $zero, $zero  # clear d
        add     $s5, $zero, $zero

        addi    $t0, $zero, 0xF    # $t2 stores the biggest 4-bit number
        add     $t3, $zero, $zero

lower:  # For 32 lower bits of each section of vector d, copy 4 lower bits in each
          element of vector a and b to each lower element of vector d
        and     $t1, $s1, $t0      # load 4 bits of $s1 (an element of a) into $t1
        and     $t2, $s3, $t0      # load 4 bits in corresponding section of $s3
                                     (an element of b) into $t2

        srlv    $t1, $t1, $t3      # shift $t1 to the proper position of the element
                                     of destination vector d
        srlv    $t2, $t2, $t3      # shift $t2 to the proper position of the element
                                     of destination vector d
```

```
        add     $s4, $s4, $t1       # copy the value stored in $t1 to vector d
        add     $s5, $s5, $t2       # copy the value stored in $t2 to vector d

        sll     $t0, $t0, 8         # adjust the pointer $t0
        addi    $t3, $t3, 4         # adjust the shift value $t3
        bne     $t0, $zero, lower   # loop until $t0 is equal to 0 after being shifted

        add     $t3, $zero, $zero   # reset the value of $t3
        addi    $t0, $zero, 0xF     # reset the value of $t0

        # For 32 higher bits of each section of vector d, copy 4 lower bits in each
          element of vector a and b to each higher element of vector d
upper:  and     $t1, $s0, $t0       # load 4 bits of $s1 (an element of a) into $t1
        and     $t2, $s2, $t0       # load 8 bits in corresponding section of $s2
                                      (an element of b) into $t2

        # shift $t1 to the proper position of the element of destination vector d
        srlv    $t1, $t1, $t3
        sll     $t1, $t1, 16

        # shift $t1 to the proper position of the element of destination vector d
        srlv    $t2, $t2, $t3
        sll     $t2, $t2, 16

        add     $s4, $s4, $t1       # copy the value stored in $t1 to vector d
        add     $s5, $s5, $t2       # copy the value stored in $t2 to vector d

        sll     $t0, $t0, 8         # adjust the pointer $t0
        addi    $t3, $t3, 4         # adjust the shift value $t3
        bne     $t0, $zero, upper   # loop until $t0 is equal to 0 after being shifted
```

```
                # Clear $t0, $t1, $t2, and $t3 after finishing the execution
                add     $t0, $zero, $zero
                add     $t1, $zero, $zero
                add     $t2, $zero, $zero
                add     $t3, $zero, $zero



                # ----------------------------------------------------------------------------
                # "Due diligence" to return control to the kernel
                # ----------------------------------------------------------------------------
exit:           ori     $v0, $zero, 10      # $v0 <-- function code for "exit"
                syscall                     # Syscall to exit



                # ****************************************************************************
                #               P R O J E C T   R E L A T E D   D A T A   S E C T I O N
                # ****************************************************************************
                .data                       # place variables, arrays, and constants, etc. in
                                              this area
```

Before:

Name	Number	Value
$zero	0	0x00000000
$at	1	0xa6570000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x5afb6c1d
$s1	17	0xae5fc041
$s2	18	0x52f3a415
$s3	19	0xa657c849
$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		0x00400024
hi		0x00000000
lo		0x00000000

After:

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	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x5afb6c1d
$s1	17	0xae5fc041
$s2	18	0x52f3a415
$s3	19	0xa657c849
$s4	20	0xabcdef01
$s5	21	0x23456789
$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		0x004000a0
hi		0x00000000
lo		0x00000000

Vector Permute

```
# ***************************341 Top Level Module*****************************
# File name:        vec_perm.asm
# Version:          1.0
# Date:             December 5, 2018
# Programmer:       Thanh Le and Steven Chung
#
# Description:      Using a sequence of MIPS instructions, create a new "SIMD
#                   Enhanced" instruction that implements a Vector Permute
#                   instruction where the syntax is vec_perm d, a, b, c. Vector a, b, c,
#                   and d consist of eight 8-bit elements. When the instruction is
#                   executed, it fills the result vector d with elements from either vector
#                   a or vector b, depending upon the "element specifier" in vector c.
#                   Each "element specifier" has two components: the most-significant
#                   half specifies an element from vector a or b (0 = a, 1 = b); the least-
#                   significant half specifies which element within the selected vector
#                   (0..7)
#
# Register usage:   $s0, $s1, $s2, $s3, $s4, $s5, $s6, $s7, $t0, $t1, $t2, $t3, $t4, $t5,
#                   $t6
#
# Notes:            $s0 and $s1 are assumed to concatenate to indicate vector a
#                   $s2 and $s3 are assumed to concatenate to indicate vector b
#                   $s4 and $s5 are assumed to concatenate to indicate vector c
#                   $s6 and $s7 are assumed to concatenate to indicate vector d
#                   $t0, $t2, and $t4 are used to store pointers
#                   $t1, $t3,$t5, and $t6 are used to store temporary data
#
#
# ******************************************************************************************
```

```
# ******************************************************************************
#                MAIN CODE SEGMENT
# ******************************************************************************
.text
.globl  main                    # main (must be global)


main:   li      $s0, 0xA567013D         # initialize a with 4 bytes from $s0 and 4
                                          bytes from $s1
        li      $s1, 0xAB45393C

        li      $s2, 0xEFC54D23         # initialize b with 4 bytes from $s2 and 4
                                          bytes from $s3
        li      $s3, 0x1277AACD

        li      $s4, 0x04171002         # initialize c with 4 bytes from $s4 and 4
                                          bytes from $s5
        li      $s5, 0x13050105

        add     $s6, $zero, $zero       # clear d
        add     $s7, $zero, $zero

        addi    $t0, $zero, 0xFF000000  # $t0, $t2, and $t4 are pointer to assist
                                          in shift and point to a specific element
        add     $t2, $zero, $zero
        addi    $t4, $zero, 24
        addi    $t3, $zero, 0xF         # $t3 stores the most biggest 4-bit
number
        addi    $t5, $zero, 8           # $t5 stores number 8
        addi    $t7, $zero, 24          # $t7 stores number 24

        # Execute the instruction on the upper 32-bit section of vector c
```

```
upperC:         and     $t1, $s4, $t0       # load 8 bits of $s4 (an element of c)
into $t1
                sllv    $t0, $t0, $t2       # shift left the pointer $t0 to the first
                                              element of the upper 32-bit section of
                                              vector c
                srlv    $t1, $t1, $t4       # shift right $t1 to the position of the last
                                              8-bit element
                slti    $t6, $t1, 0x10      # compare the value with 0x10 to
                                              determine if it takes the data from
                                              vector a or vector b
                and     $t1, $t1, $t3       # load the lower 4 bits in the 8-bit
                                              element to $t1
                bne     $t6, $zero, takeA   # choose vector if the upper 4 bits has
                                              the value of 0

                # Determine if the index points to the element in upper 32-bit section or
                  lower 32-bit section of vector b
takeB:          slti    $t6, $t1, 4
                bne     $t6, $zero, takeUpperB

                # Determine the element at the index if it is in the lower 32-bit section of
                  vector b
takeLowerB:     mult    $t1, $t5
                mflo    $t6
                addi    $t6, $t6, -32
                srlv    $t0, $t0, $t6
                and     $t1, $s3, $t0
                j       execute

                # Determine element at the index if it is in the upper 32-bit section of
                  vector b
takeUpperB:     mult    $t1, $t5
                mflo    $t6
```

```
                srlv    $t0, $t0, $t6
                and     $t1, $s2, $t0
                j       execute

                # Determine if the index points to the element in upper 32-bit section or
                  lower 32-bit section of vector a
takeA:          slti    $t6, $t1, 4
                bne     $t6, $zero, takeUpperA

                # Determine the element at the index if it is in the upper 32-bit section of
                  vector a
takeLowerA:     mult    $t1, $t5
                mflo    $t6
                addi    $t6, $t6, -32
                srlv    $t0, $t0, $t6
                and     $t1, $s1, $t0
                j       execute

                # Determine element at the index if it is in the lower 32-bit section of
                  vector b
takeUpperA:     mult    $t1, $t5
                mflo    $t6
                srlv    $t0, $t0, $t6
                and     $t1, $s0, $t0

execute:        sllv    $t0, $t0, $t6       # shift the $t0 to its previous position
                srlv    $t0, $t0, $t2
                sub     $t6, $t7, $t6       # calculate the bits to be shifted to move the
element to the last index
                srlv    $t1, $t1, $t6       # shift right the element to the last index
```

```
            sllv    $s6, $s6, $t5       # shift left the $s6 8 bits to give space for the
                                          element at the last index
            add     $s6, $s6, $t1       # copy the elemment to the end of $s6

            # Adjust the pointers
            srl     $t0, $t0, 8
            addi    $t2, $t2, 8
            sub     $t4, $t4, 8
            bne     $t0, $zero, upperC

            #-------------------------------------------------------------------------------------
            # Repeat the logic for the lower 32-bit section of vector c after resetting
              $t0, $t2, $t3, $t4, $t5, and $t7
            addi    $t0, $zero, 0xFF000000
            add     $t2, $zero, $zero
            addi    $t3, $zero, 0xF
            addi    $t4, $zero, 24
            addi    $t5, $zero, 8
            addi    $t7, $zero, 24

lowerC:     and     $t1, $s5, $t0
            sllv    $t0, $t0, $t2

            srlv    $t1, $t1, $t4

            slti    $t6, $t1, 0x10
            and     $t1, $t1, $t3
            bne     $t6, $zero, takeA2

takeB2:     slti    $t6, $t1, 4
            bne     $t6, $zero, takeUpperB2
```

```
takeLowerB2: mult  $t1, $t5
             mflo  $t6
             addi  $t6, $t6, -32
             srlv  $t0, $t0, $t6
             and   $t1, $s3, $t0
             j     execute2

takeUpperB2: mult  $t1, $t5
             mflo  $t6
             srlv  $t0, $t0, $t6
             and   $t1, $s2, $t0
             j     execute2

takeA2:      slti  $t6, $t1, 4
             bne   $t6, $zero, takeUpperA2

takeLowerA2: mult  $t1, $t5
             mflo  $t6
             addi  $t6, $t6, -32
             srlv  $t0, $t0, $t6
             and   $t1, $s1, $t0
             j     execute2

takeUpperA2: mult  $t1, $t5
             mflo  $t6
             srlv  $t0, $t0, $t6
             and   $t1, $s0, $t0

execute2:    sllv  $t0, $t0, $t6
             srlv  $t0, $t0, $t2
            sub    $t6, $t7, $t6
```

```
            srlv    $t1, $t1, $t6

            sllv    $s7, $s7, $t5
            add     $s7, $s7, $t1

            srl     $t0, $t0, 8
            addi    $t2, $t2, 8
            sub     $t4, $t4, 8
            bne     $t0, $zero, lowerC

            # Clear $t0, $t1, $t2, and $t3 after finishing the execution
            add     $t0, $zero, $zero
            add     $t1, $zero, $zero
            add     $t2, $zero, $zero
            add     $t3, $zero, $zero
            add     $t4, $zero, $zero
            add     $t5, $zero, $zero
            add     $t6, $zero, $zero

            # ---------------------------------------------------------------------------
            # "Due diligence" to return control to the kernel
            # ---------------------------------------------------------------------------
exit:       ori     $v0, $zero, 10      # $v0 <-- function code for "exit"
            syscall                     # Syscall to exit


            # ***************************************************************************
            #             P R O J E C T   R E L A T E D   D A T A   S E C T I O N
            # ***************************************************************************
            .data                       # place variables, arrays, and constants, etc. in
                                          this area
```

Before:

Name	Number	Value
$zero	0	0x00000000
$at	1	0x13050000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0xa567013d
$s1	17	0xab45393c
$s2	18	0xefc54d23
$s3	19	0x1277aacd
$s4	20	0x04171002
$s5	21	0x13050105
$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		0x00400038
hi		0x00000000
lo		0x00000000

After:

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000008
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000018
$s0	16	0xa567013d
$s1	17	0xab45393c
$s2	18	0xefc54d23
$s3	19	0x1277aacd
$s4	20	0x04171002
$s5	21	0x13050105
$s6	22	0xabcdef01
$s7	23	0x23456745
$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		0x004001ec
hi		0x00000000
lo		0x00000028

Vector Compare Equal-To

```
# ***************************3 4 1  T o p  L e v e l  M o d u l e ******************************
# File name:          vec_cmpeq.asm
# Version:            1.0
# Date:               December 5, 2018
# Programmer:         Thanh Le and Steven Chung
#
# Description:        Using a sequence of MIPS instructions, create a new "SIMD
#                     Enhanced" instruction that implements a Vector Compare
#                      Equal-To instruction where the syntax is vec_cmpeq d, a, b.
#                      Vector a, b, and d consist of eight 8-bit elements. When the
#                      instruction is execute, each element of the result vector d is
#                     TRUE (all bits = 1) if the corresponding element of vector a is
#                      equal to the corresponding element of vector b. Otherwise, the.
#                      element of result is FALSE (all bits = 0)
#
# Register usage:     $s0, $s1, $s2, $s3, $s4, $s5, $t0, $t1
#
# Notes:              $s0 and $s1 are assumed to concatenate to indicate vector a
#                     $s2 and $s3 are assumed to concatenate to indicate vector b
#                     $s4 and $s5 are assumed to concatenate to indicate vector d
#                     $t0 is used to point to a specific elements or bits
#                     $t1 and $t2 are used to store temporary data
#
# ********************************************************************************************


                # ********************************************************************************
                #                 M A I N   C O D E   S E G M E N T
                # ********************************************************************************
                .text
                .globl  main                        # main (must be global)
```

```
main:       li      $s0, 0x5AFB6C1D     # initialize a with 4 bytes from $s0 and 4
                                          bytes from $s1
            li      $s1, 0xA65FC040

            li      $s2, 0x52FBA415     # initialize b with 4 bytes from $s2 and 4
                                          bytes from $s3
            li      $s3, 0xAE5FC841

            add     $s4, $zero, $zero   # clear d
            add     $s5, $zero, $zero

            addi    $t0, $zero, 0xFF

loop:       and     $t1, $s0, $t0       # load 8 bits of $s0 (an element of a)
                                          into $t1
            and     $t2, $s2, $t0       # load 8 bits in corresponding section of
                                          $s2 (an element of b) into $t2
            bne     $t1, $t2, notEqual1 # jump to notEqual1 if they are not equal

equal1:     add     $s4, $s4, $t0       # load 8 bits in corresponding section of
                                          $s2 (an element of b) into $t2

notEqual1:  and     $t1, $s1, $t0       # load 8 bits of $s0 (an element of a)
                                          into $t1
            and     $t2, $s3, $t0       # load 8 bits in corresponding section of
                                          $s3 (an element of b) into $t2
            bne     $t1, $t2, notEqual2 # jump to notEqual2 if they are not equal

equal2:     add     $s5, $s5, $t0       # load 8 bits in corresponding section of
                                          $s2 (an element of b) into $t2

notEqual2:  sll     $t0, $t0, 8         # adjust the pointer $t0 to the next
                                          element
```

```
        bne     $t0, $zero, loop            # loop until the pointer $t0 is equal to 0

        # Clear $t0, $t1, and $t2 after finishing the execution
        add     $t0, $zero, $zero
        add     $t1, $zero, $zero
        add     $t2, $zero, $zero

        # ---------------------------------------------------------------------------------------
        # "Due diligence" to return control to the kernel
        # ---------------------------------------------------------------------------------------
exit:   ori             $v0, $zero, 10      # $v0 <-- function code for "exit"
        syscall                             # Syscall to exit


        # ***************************************************************************
        #               P R O J E C T   R E L A T E D   D A T A   S E C T I O N
        # ***************************************************************************
        .data                               # place variables, arrays, and constants,
                                              etc. in this area
```

Before:

Name	Number	Value
$zero	0	0x00000000
$at	1	0xae5f0000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x5afb6c1d
$s1	17	0xa65fc040
$s2	18	0x52fba415
$s3	19	0xae5fc841
$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		0x00400028
hi		0x00000000
lo		0x00000000

After:

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	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x5afb6c1d
$s1	17	0xa65fc040
$s2	18	0x52fba415
$s3	19	0xae5fc841
$s4	20	0x00ff0000
$s5	21	0x00ff0000
$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		0x00400068
hi		0x00000000
lo		0x00000000

Vector Compare Less-Than (unsigned)

```
# ***************************341 Top Level Module*****************************

# File name:            vec_cmpltu.asm
# Version:              1.0
# Date:                 December 5, 2018
# Programmer:           Thanh Le and Steven Chung
#
# Description:          Using a sequence of MIPS instructions, create a
#                       new “SIMD Enhanced" instruction that implements a
#                       Vector Compare Less-Than (unsigned) instruction
#                       where the syntax is vec_cmpltu d, a, b. Vector a, b,
#                       and d consist of eight 8-bit elements. When the
#                       instruction is execute, each element of the result
#                       vector d is TRUE (all bits = 1) if the corresponding
#                       element of vector a is less than the corresponding
#                       element of vector b. Otherwise, the element of result is
#                       FALSE (all bits = 0)
#
# Register usage:       $s0, $s1, $s2, $s3, $s4, $s5, $t0, $t1
#
# Notes:                $s0 and $s1 are assumed to concatenate to
#                       indicate vector a
#                       $s2 and $s3 are assumed to concatenate to
#                       indicate vector b
#                       $s4 and $s5 are assumed to concatenate to
#                       indicate vector d
#                       $t0 is used to point to a specific elements or bits
#                       $t1 and $t2 are used to store temporary data
#
# ******************************************************************************************
```

```
# ******************************************************************************
#                MAIN CODE SEGMENT
# ******************************************************************************
.text
.globl  main                            # main (must be
                                        global)

main:   li      $s0, 0x5AFB6C1D         # initialize a with 4
                                        bytes from $s0 and 4
                                        bytes from $s1
        li      $s1, 0xA65FC040

        li      $s2, 0x52FBA415         # initialize b with 4 bytes
                                        from $s2  and 4 bytes from
                                        $s3
        li      $s3, 0xAE5FC841

        add     $s4, $zero, $zero       # clear d
        add     $s5, $zero, $zero

        addi    $t0, $zero, 0xFF

loop:   and     $t1, $s0, $t0           # load 8 bits of $s0 (an
                                        element of a) into $t1
        and     $t2, $s2, $t0           # load 8 bits in
                                        corresponding section of $s2
                                        (an element of b) into $t2
        slt     $t1, $t1, $t2           # check if $t1 is less than $t2
        beq     $t1, $zero, notLessThan1    # jump to notLessThan1 if
                                        $t1 is not less than $t2
```

```
lessThan1:    add     $s4, $s4, $t0                   # load 8 bits in
                                                      corresponding
                                                      section of $s2 (an element of
                                                      b)into $t2

notLessThan1:         and     $t1, $s1, $t0           # load 8 bits of $s0 (an
                                                      element of a) into $t1
                      and     $t2, $s3, $t0           # load 8 bits in
                                                      corresponding section of $s3
                                                      (an element of b) into $t2
                      slt     $t1, $t1, $t2           # check if $t1 is less than $t2
                      beq     $t1, $zero, notLessThan2 # jump to
                                                      notLessThan1 if $t1 is
                                                      not less than $t2

lessThan2:            add     $s5, $s5, $t0           # load 8 bits in
                                                      corresponding section of $s2 (an
                                                      element of b) into $t2

notLessThan2:         sll     $t0, $t0, 8             # adjust the pointer $t0 to the
                                                      next element
                      bne     $t0, $zero, loop        # loop until the pointer $t0 is
                                                      equal to 0

                # Clear $t0, $t1, and $t2 after finishing the execution
                add     $t0, $zero, $zero
                add     $t1, $zero, $zero
                add     $t2, $zero, $zero

                # ------------------------------------------------------------------------
                # "Due diligence" to return control to the kernel
                # ------------------------------------------------------------------------
exit:           ori     $v0, $zero, 10              # $v0 <-- function code for "exit"
```

```
syscall                         # Syscall to exit


# ******************************************************************************
#               P R O J E C T   R E L A T E D   D A T A   S E C T I O N
# ******************************************************************************
.data                           # place variables, arrays, and constants,
                                etc. in this area
```

Before:

Name	Number	Value
$zero	0	0x00000000
$at	1	0xae5f0000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x5afb6c1d
$s1	17	0xa65fc040
$s2	18	0x52fba415
$s3	19	0xae5fc841
$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		0x00400028
hi		0x00000000
lo		0x00000000

After:

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	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x5afb6c1d
$s1	17	0xa65fc040
$s2	18	0x52fba415
$s3	19	0xae5fc841
$s4	20	0x0000ff00
$s5	21	0xff00ffff
$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		0x00400070
hi		0x00000000
lo		0x00000000

Vector Shift Right Arithmetically

```
# ***************************341 Top Level Module*****************************
# File name:        vec_sra.asm
# Version:          1.0
# Date:             December 5, 2018
# Programmer:       Thanh Le and Steven Chung
#
# Description:      Using a sequence of MIPS instructions, create a new "SIMD
#                   Enhanced" instruction that implements a Vector Shift Right
#                   Arithmetically instruction where the syntax is vec_sra d, a, b. Vector
#                   a and d consist of eight 8-bit elements. When the instruction is
#                   executed, vector a is shifted right arithmetically by the number of
#                   elements stored in b.
#
# Register usage:   $s0, $s1, $s2, $s3, $s4, $t0, $t1, $t2, $t3
#
# Notes:            $s0 and $s1 are assumed to concatenate to indicate vector a
#                   $s3 and $s4 are assumed to concatenate to indicate vector d
#                   $s2 is assumed to indicate the number of 8-bit element to be
#                   shifted
#                   $t0, $t1, $t2, and $t3 are used to store temporary data
#
# ********************************************************************************************
```

```
                # ******************************************************************************
                #                MAIN CODE SEGMENT
                # ******************************************************************************
                .text
                .globl  main                    # main (must be global)


main:           li      $s0, 0xABCDEF01         # initialize a with 4 bytes from $s0 and 4
                                                  bytes from $s1
                li      $s1, 0x23456789

                li      $s2, 2                  # initialize the number of bits to be
                                                  shifted

                add     $s3, $zero, $zero       # clear d
                add     $s4, $zero, $zero

                addi    $t0, $zero, 0xF0000000
                addi    $t3, $zero, 8           # store the value of 8 in $t4

                mult    $s2, $t3                # calculate the number of bits to be
                                                  shifted
                mflo    $t1                     # store the number of bits to be shifted
                                                  in $t1
                slti    $t2, $t1, 32            # compare if the number of shifted bits
                                                  is less than 32
                bne     $t2, $zero, upper       # jump to upper if the number of shifted
                                                  is less than 32

                # If the number of bits to be shifted is greater or equal to 32
lower:          addi    $t1, $t1, -32           # calculate the number of bits to be
                                                  shifted in lower 32-bit section
```

```
            srav    $s4, $s0, $t1           # shift right arithmetically the lower 32-
                                              bit section of vector a
            and     $t1, $s4, $t0           # take the first number in the section
            bne     $t1, $t0, clearReg      # if the first number is 0, the upper 32-bit
                                              section can be left as 0x0
            addi    $s3, $s3, 0xFFFFFFFF    # if the first number is F, set the upper
                                              32-bit section to 0xFFFFFFF
            j       clearReg

            # If the number of bits to be shifted is less than 32
upper:      srav    $s3, $s0, $t1           # shift right arithmetically the upper 32-
                                              bit section of vector a
                                            # and store the result in the upper 32-
                                              bit section of vector d
            sllv    $t2, $s0, $t1           # take the bits of the upper section that
                                              are truncated after being shifted
            srlv    $s4, $s1, $t1           # shift right the lower 32-bit section of
                                              vector a by same number of bits
                                            # that is shifted in the upper section to
                                              leave space for the missed bits
            add     $s4, $s4, $t2           # copy the missed bits to the lower 32-
                                              bit section of vector d

            # Clear $t0, $t1, and $t2 after finishing the execution
clearReg:   add     $t0, $zero, $zero
            add     $t1, $zero, $zero
            add     $t2, $zero, $zero
            add     $t3, $zero, $zero
```

```
                # ---------------------------------------------------------------------------
                # "Due diligence" to return control to the kernel
                # ---------------------------------------------------------------------------
exit:           ori     $v0, $zero, 10          # $v0 <-- function code for "exit"
                syscall                         # Syscall to exit



                # ***************************************************************************
                #               P R O J E C T   R E L A T E D   D A T A   S E C T I O N
                # ***************************************************************************
                .data                           # place variables, arrays, and constants,
                                                  etc. in this area
```

Before:

Name	Number	Value
$zero	0	0x00000000
$at	1	0x12340000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0xabcdef01
$s1	17	0x12345678
$s2	18	0x00000002
$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		0x00400018
hi		0x00000000
lo		0x00000000

After:

Name	Number	Value
$zero	0	0x00000000
$at	1	0xf0000000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000008
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0xabcdef01
$s1	17	0x12345678
$s2	18	0x00000002
$s3	19	0xffffabcd
$s4	20	0xef011234
$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		0x00400080
hi		0x00000000
lo		0x00000010

Vector Shift Right Logically

```
# ***************************341 Top Level Module*****************************
# File name:        vec_srl.asm
# Version:          1.0
# Date:             December 5, 2018
# Programmer:       Thanh Le and Steven Chung
#
# Description:      Using a sequence of MIPS instructions, create a new "SIMD
#                   Enhanced" instruction that implements a Vector Shift Right
#                   Logically instruction where the syntax is vec_srl d, a, b. Vector a
#                   executed, vector a is shifted right logically by the number of
#                   elements stored in b.
#
# Register usage:   $s0, $s1, $s2, $s3, $s4, $t0, $t1, $t2
#
# Notes:            $s0 and $s1 are assumed to concatenate to indicate vector a
#                   $s3 and $s4 are assumed to concatenate to indicate vector d
#                   $s2 is assumed to indicate the number of 8-bit element to be
#                   shifted
#                   $t0, $t1, and $t2 are used to store temporary data
#
# ******************************************************************************************
```

```
                # ******************************************************************************
                #                MAIN CODE SEGMENT
                # ******************************************************************************
                .text
                .globl  main                    # main (must be global)


main:           li      $s0, 0xABCDEF01         # initialize a with 4 bytes from $s0 and 4
                                                  bytes from $s1
                li      $s1, 0x23456789

                li      $s2, 2                  # initialize the number of bits to be
                                                  shifted

                add     $s3, $zero, $zero       # clear d
                add     $s4, $zero, $zero

                addi    $t0, $zero, 8           # store the value of 8 in $t4

                mult    $s2, $t0                # calculate the number of bits to be
                                                  shifted
                mflo    $t1                     # store the number of bits to be shifted
                                                  in $t1
                slti    $t2, $t1, 32            # compare if the number of shifted bits
                                                  is less than 32
                bne     $t2, $zero, upper       # jump to upper if the number of shifted
                                                  is less than 32

                # If the number of bits to be shifted is greater or equal to 32
lower:          addi    $t1, $t1, -32           # calculate the number of bits to be
                                                  shifted in lower 32-bit section
                srlv    $s4, $s0, $t1           # shift right logically the lower 32-bit
                                                  section of vector a
```

```
                j       clearReg

                # If the number of bits to be shifted is less than 32
upper:          srlv    $s3, $s0, $t1           # shift right the upper 32-bit section of
                                                  vector a
                                                # and store the result in the upper 32-bit
                                                  section of vector d
                sllv    $t2, $s0, $t1           # take the bits of the upper section that
                                                  are truncated after being shifted
                srlv    $s4, $s1, $t1           # shift left the lower 32-bit section of
                                                  vector a by same number of bits
                                                # that is shifted in the upper section to
                                                  leave space for the missed bits
                add     $s4, $s4, $t2           # copy the missed bits to the lower 32-
                                                  bit section of vector d

                # Clear $t0, $t1, and $t2 after finishing the execution
clearReg:       add     $t0, $zero, $zero
                add     $t1, $zero, $zero
                add     $t2, $zero, $zero

                # -----------------------------------------------------------------------------
                # "Due diligence" to return control to the kernel
                # -----------------------------------------------------------------------------
exit:           ori     $v0, $zero, 10          # $v0 <-- function code for "exit"
                syscall                         # Syscall to exit

                # *****************************************************************************
                #               P R O J E C T   R E L A T E D   D A T A   S E C T I O N
                # *****************************************************************************
                .data                           # place variables, arrays, and constants,
                                                  etc. in this area
```

Before:

Name	Number	Value
$zero	0	0x00000000
$at	1	0x23450000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0xabcdef01
$s1	17	0x23456789
$s2	18	0x00000002
$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		0x00400018
hi		0x00000000
lo		0x00000000

After:

Name	Number	Value
$zero	0	0x00000000
$at	1	0x23450000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0xabcdef01
$s1	17	0x23456789
$s2	18	0x00000002
$s3	19	0x0000abcd
$s4	20	0xef012345
$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		0x00400060
hi		0x00000000
lo		0x00000010

Vector Shift Left Logically

```
# ***************************341 Top Level Module*****************************
# File name:        vec_sll.asm
# Version:          1.0
# Date:             December 5, 2018
# Programmer:       Thanh Le and Steven Chung
#
# Description:      Using a sequence of MIPS instructions, create a new "SIMD
#                   Enhanced" instruction that implements a Vector Shift Left Logically
#                   instruction where the syntax is vec_sll d, a, b. Vector a and d
#                   consist of eight 8-bit elements. When the instruction is executed,
#                   vector a is shifted left logically by the number of elements stored in
#                   b.
#
# Register usage:   $s0, $s1, $s2, $s3, $s4, $t0, $t1, $t2
#
# Notes:            $s0 and $s1 are assumed to concatenate to indicate vector a
#                   $s3 and $s4 are assumed to concatenate to indicate vector d
#                   $s2 is assumed to indicate the number of 8-bit element to be
#                   shifted
#                   $t0, $t1, and $t2 are used to store temporary data
#
# ******************************************************************************************
```

```
# ******************************************************************************
#                MAIN CODE SEGMENT
# ******************************************************************************
.text
.globl  main                    # main (must be global)


main:   li      $s0, 0xABCDEF01     # initialize a with 4 bytes from $s0 and 4
                                      bytes from $s1
        li      $s1, 0x23456789

        li      $s2, 2              # initialize the number of bits to be
                                      shifted

        add     $s3, $zero, $zero   # clear d
        add     $s4, $zero, $zero

        addi    $t0, $zero, 8       # store the value of 8 in $t4

        mult    $s2, $t0            # calculate the number of bits to be
                                      shifted
        mflo    $t1                 # store the number of bits to be shifted
                                      in $t1
        slti    $t2, $t1, 32        # compare if the number of shifted bits
                                      is less than 32
        bne     $t2, $zero, upper   # jump to upper if the number of shifted
                                      is less than 32

        # If the number of bits to be shifted is greater or equal to 32
lower:  addi    $t1, $t1, -32       # calculate the number of bits to be
                                      shifted in lower 32-bit section
        sllv    $s4, $s0, $t1       # shift left logically the lower 32-bit
                                      section of vector a
```

```
                j       clearReg

                # If the number of bits to be shifted is less than 32
upper:          sllv    $s3, $s0, $t1           # shift left the upper 32-bit section of
                                                  vector a and store the result in the
                                                  upper 32- bit section of vector d
                srlv    $t2, $s1, $t1           # take the bits of the lower section that
                                                  are truncated after being shifted
                sllv    $s4, $s1, $t1           # shift left the lower 32-bit section of
                                                  vector a by same number of bits that
                                                  is shifted in the upper section to leave
                                                  space for the missed bits
                add     $s3, $s3, $t2           # copy the missed bits to the upper 32-
                                                  bit section of vector d

                # Clear $t0, $t1, and $t2 after finishing the execution
clearReg:       add     $t0, $zero, $zero
                add     $t1, $zero, $zero
                add     $t2, $zero, $zero



                # ---------------------------------------------------------------------------
                # "Due diligence" to return control to the kernel
                # ---------------------------------------------------------------------------
exit:           ori     $v0, $zero, 10          # $v0 <-- function code for "exit"
                syscall                         # Syscall to exit



                # ***************************************************************************
                #               P R O J E C T   R E L A T E D   D A T A   S E C T I O N
                # ***************************************************************************
                .data                           # place variables, arrays, and constants,
                                                  etc. in this area
```

Before:

Name	Number	Value
$zero	0	0x00000000
$at	1	0x23450000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0xabcdef01
$s1	17	0x23456789
$s2	18	0x00000002
$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		0x00400018
hi		0x00000000
lo		0x00000000

After:

Name	Number	Value
$zero	0	0x00000000
$at	1	0x23450000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0xabcdef01
$s1	17	0x23456789
$s2	18	0x00000002
$s3	19	0xef012345
$s4	20	0x67890000
$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		0x00400060
hi		0x00000000
lo		0x00000010

Vector Replace All

```
# ***************************341 Top Level Module*****************************
# File name:        vec_repl.asm
# Version:          1.0
# Date:             December 5, 2018
# Programmer:       Thanh Le and Steven Chung
#
# Description:      Using a sequence of MIPS instructions, create a new "SIMD
#                   Enhanced" instruction that implements a Vector Replace All
#                   instruction where the syntax is vec_repl d, a, b. Vector a and d
#                   consist of eight 8-bit elements. Component b is a 8-bit number in
#                   hexadecimal. When the instruction is executed, all the 8-bit
#                   elements in vector a are replaced by component b. The result
#                  result is stored in vector d.
#
# Register usage:   $s0, $s1, $s2, $s3, $s4, $t0
#
# Notes:            $s0 and $s1 are assumed to concatenate to indicate vector a
#                   #s2 is used to indicate the 8-bit hexadecimal number of component
#                   b
#                   $s3 and $s4 are assumed to concatenate to indicate vector d
#                   $t0 is used to store temporary data
#
# ******************************************************************************************
```

```
# ******************************************************************************
#                MAIN CODE SEGMENT
# ******************************************************************************
.text
.globl  main                    # main (must be global)


main:   li      $s0, 0xAABBCCDD     # initialize a with 4 bytes from $s0 and 4
                                      bytes from $s1
        li      $s1, 0x11223344

        li      $s2, 0xE8           # initialize b

        add     $s3, $zero, $zero   # clear d
        add     $s4, $zero, $zero

        add     $t0, $zero, $s2     # copy the original value of component b
                                      to $t0

loop:   add     $s3, $s3, $t0       # copy the value of component b to an
                                      element in the upper 32-bit section of
                                      vector d
        add     $s4, $s4, $t0       # copy the value of component b to an
                                      element in the lower 32-bit section of
                                      vector d

        sll     $t0, $t0, 8         # shift the $t0 left to the next element
        bne     $t0, $zero, loop    # loop until $t0 is shifted all to the left
                                      and is equal to 0
```

```
                # Clear $t0, $t1, and $t2 after finishing the execution
                add     $t0, $zero, $zero

                # -----------------------------------------------------------------------------
                # "Due diligence" to return control to the kernel
                # -----------------------------------------------------------------------------
exit:           ori     $v0, $zero, 10          # $v0 <-- function code for "exit"
                syscall                         # Syscall to exit


                # *****************************************************************************
                #               P R O J E C T   R E L A T E D   D A T A   S E C T I O N
                # *****************************************************************************
                .data                           # place variables, arrays, and constants,
                                                  etc. in this area
```

Before:

Name	Number	Value
$zero	0	0x00000000
$at	1	0x11220000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0xaabbccdd
$s1	17	0x11223344
$s2	18	0x000000e8
$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		0x00400014
hi		0x00000000
lo		0x00000000

After:

Name	Number	Value
$zero	0	0x00000000
$at	1	0x11220000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0xaabbccdd
$s1	17	0x11223344
$s2	18	0x000000e8
$s3	19	0xe8e8e8e8
$s4	20	0xe8e8e8e8
$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		0x0040003c
hi		0x00000000
lo		0x00000000

Vector Reverse

```
# ***************************341 Top Level Module*****************************
# File name:        vec_rev.asm
# Version:          1.0
# Date:             December 5, 2018
# Programmer:       Thanh Le and Steven Chung
#
# Description:      Using a sequence of MIPS instructions, create a new "SIMD
#                   Enhanced" instruction that implements a Vector Reverse instruction
#                   where the syntax is vec_rev d, a. Vector a and d consist of eight 8-
#                   bit elements. When the instruction is executed, it reverses the order
#                   of the elements in vector a and stores the change in vector d
#
# Register usage:   $s0, $s1, $s2, $s3, $t0, $t1, $t2, $t3, $t4
#
# Notes:            $s0 and $s1 are assumed to concatenate to indicate vector a
#                   $s2 and $s3 are assumed to concatenate to indicate vector d
#                   $t0, $t1, $t2, $t3, and $t4 are used to store temporary data
#
# ******************************************************************************************
```

```
            # ******************************************************************************
            #                MAIN CODE SEGMENT
            # ******************************************************************************
            .text
            .globl  main                    # main (must be global)


main:       li      $s0, 0x11223344         # initialize a with 4 bytes from $s0 and 4
                                              bytes from $s1
            li      $s1, 0x55667788

            addi    $t0, $t0, 0xFF000000
            addi    $t3, $t3, 0xFF
            addi    $t4, $t4, 24

            add     $s3, $zero, $zero       # clear d
            add     $s4, $zero, $zero

loop:       and     $t1, $s0, $t0           # load 8-bit element of upper section of
                                              vector a into $t1
            ror     $t2, $t1, $t4           # move the 8-bit element to its reversed
                                              position and store in $t2
            and     $t1, $s1, $t3           # load 8-bit element of lower section of
                                              vector a into $t1
            rol     $t1, $t1, $t4           # move the 8-bit element to its reversed
                                              position and store in $t1

            # copy these elements to the destination vector d
            add     $s2, $s2, $t1
            add     $s3, $s3, $t2
```

```
                # adjust the pointers
                srl     $t0, $t0, 8
                sll     $t3, $t3, 8
                addi    $t4, $t4, 16
                bne     $t0, $zero, loop            # loop until $t0 is equal to 0

                # Clear $t0, $t1, and $t2 after finishing the execution
                add     $t0, $zero, $zero
                add     $t1, $zero, $zero
                add     $t2, $zero, $zero
                add     $t3, $zero, $zero
                add     $t4, $zero, $zero

                # ---------------------------------------------------------------------------
                # "Due diligence" to return control to the kernel
                # ---------------------------------------------------------------------------
exit:           ori     $v0, $zero, 10              # $v0 <-- function code for "exit"
                syscall                             # Syscall to exit


                # ***************************************************************************
                #               P R O J E C T   R E L A T E D   D A T A   S E C T I O N
                # ***************************************************************************
                .data                               # place variables, arrays, and constants,
                                                      etc. in this area
```

Before:

Name	Number	Value
$zero	0	0x00000000
$at	1	0x55660000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x11223344
$s1	17	0x55667788
$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		0x00400010
hi		0x00000000
lo		0x00000000

After:

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000055
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x11223344
$s1	17	0x55667788
$s2	18	0x88776655
$s3	19	0x44332211
$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		0x00400088
hi		0x00000000
lo		0x00000000

Vector Swap Element

```
# ***************************341 Top Level Module*****************************
# File name:        vec_swap_el.asm
# Version:          1.0
# Date:             December 5, 2018
# Programmer:       Thanh Le and Steven Chung
#
# Description:      Using a sequence of MIPS instructions, create a new "SIMD
#                   Enhanced" instruction that implements a Vector Replace All
#                   instruction where the syntax is vec_swap_el d, a, b, c. Vector a and
#                   d consist of eight 8-bit elements. Component b and d indicate the
#                   index of 2 elements to be swapped. When the instruction is
#                   executed, the element at index b is swapped with the element at
#                   index c of vector a. The result is stored in vector d.
#
# Register usage:   $s0, $s1, $s2, $s3, $s4, $s5, $t0, $t1, $t2, $t3, $t4, $t5, $t6, $v0,
#                   $v1
#
# Notes:            $s0 and $s1 are assumed to concatenate to indicate vector a
#                   #s2 and $s3 are used to indicate indexes of the elements to be
#                   swapped
#                   $s3 and $s4 are assumed to concatenate to indicate vector d
#                   $v0 and $v1 are used to store temporary data of destination vector
#                   d.
#                   $t0, $t1, $t2, $t3, $t4, $t5, and $t6 are used to store temporary data
#
# ******************************************************************************************
```

```
# ******************************************************************************
#                MAIN CODE SEGMENT
# ******************************************************************************
.text
.globl  main                    # main (must be global)


main:   li      $s0, 0xAABBCCDD     # initialize a with 4 bytes from $s0 and 4
                                      bytes from $s1
        li      $s1, 0x11223344

        li      $s2, 1              # initialize the one of the two indexes
        li      $s3, 4              # initialize the other index

        add     $s4, $zero, $zero   # clear d
        add     $s5, $zero, $zero

        add     $t0, $zero, 0xFF000000
        addi    $t1, $zero, 8
        addi    $t4, $zero, 24

        add     $v0, $zero, $s0     # copy the upper 32-bit section of vector
                                      a to $v0
        add     $v1, $zero, $s1     # copy the lower 32-bit section of vector
                                      a to $v1

        # Determine the element at the first index, move it to the position of the
          other index, and delete the proper position for the element at the other to
          be swapped in
        mult    $s2, $t1            # calculate the number of bits the
                                      pointer $t0 to be shifted
                                    # to point to the first index
```

```
            mflo    $t2

            mult    $s3, $t1            # calculate the number of bits the
                                          pointer $t0 to be shifted to point to the
                                          second index
            mflo    $t3

            slti    $t5, $t3, 32        # compare to determine if the index is in
                                          upper or lower section
            bne     $t5, $zero, noSub1
            addi    $t3, $t3, -32

noSub1:     slti    $t5, $t2, 32        # compare to determine if the index is in
                                          upper or lower section
            bne     $t5, $zero, upper1

            # If the index points to an element in the lower section
lower1:         addi    $t2, $t2, -32   # recalculate the number of bits to be
                                          shifted in lower section to reach the
                                          index
            srlv    $t0, $t0, $t2       # shift the pointer to the index
            or      $v1, $v1, $t0       # replace the element at the index with
                                          FF
            xor     $v1, $v1, $t0       # replace the element at the index with
                                          00
            and     $t5, $s1, $t0       # take the element at the index
            j       jump1

upper1:     srlv    $t0, $t0, $t2       # shift the pointer to the index
            or      $v0, $v0, $t0       # replace the element at the index with
                                          FF
            xor     $v0, $v0, $t0       # replace the element at the index with
                                          00
```

```
                and     $t5, $s0, $t0           # take the element at the index

jump1:          sub     $t2, $t4, $t2           # calculate the number of bits to be
                                                  shifted to move the element to the
                                                  most right position
                srlv    $t5, $t5, $t2           # shift the element to the most right
                                                  position
                sub     $t2, $t4, $t3           # calculate the number of bits to be
                                                  shifted to move the element to the
                                                  swapped position
                sllv    $t5, $t5, $t2           # shift the element to the swapped
                                                  position

#-------------------------------------------------------------------------------------------------------------
                # Reset the pointer $t0 and repeat the same logic for the other index
                add     $t0, $zero, 0xFF000000

                mult    $s3, $t1
                mflo    $t3

                mult    $s2, $t1
                mflo    $t2

                slti    $t6, $t2, 32
                bne     $t6, $zero, noSub2
                addi    $t2, $t2, -32

noSub2:         slti    $t6, $t3, 32
                bne     $t6, $zero, upper2

lower2:         addi    $t3, $t3, -32
                srlv    $t0, $t0, $t3
```

```
                or      $v1, $v1, $t0
                xor     $v1, $v1, $t0
                and     $t6, $s1, $t0
                j       jump2

upper2:         srlv    $t0, $t0, $t3
                or      $v0, $v0, $t0
                xor     $v0, $v0, $t0
                and     $t6, $s0, $t0

jump2:          sub     $t3, $t4, $t3
                srlv    $t6, $t6, $t3
                sub     $t3, $t4, $t2
                sllv    $t6, $t6, $t3


#-------------------------------------------------------------------------------------------------------------
                # Swap the two elements at two indicated indexes
                mult    $s2, $t1
                mflo    $t2
                slti    $t2, $t2, 32
                bne     $t2, $zero, lessThan1
                add     $s5, $v1, $t6
                j       clearReg

lessThan1:      add     $s4, $v0, $t6

next:           mult    $s3, $t1
                mflo    $t3
                slti    $t3, $t3, 32
                bne     $t3, $zero, lessThan2
```

```
                add     $s5, $v1, $t5
                j       clearReg

lessThan2:      add     $s4, $v0, $t5


                # Clear $t0, $t1, $t2, $t3, $t4, $t5, $t6, $v0, and $v1 after finishing the
                  execution
clearReg:       add     $t0, $zero, $zero
                add     $t1, $zero, $zero
                add     $t2, $zero, $zero
                add     $t3, $zero, $zero
                add     $t4, $zero, $zero
                add     $t5, $zero, $zero
                add     $t6, $zero, $zero
                add     $v0, $zero, $zero
                add     $v1, $zero, $zero


                # ---------------------------------------------------------------------------
                # "Due diligence" to return control to the kernel
                # ---------------------------------------------------------------------------
exit:           ori     $v0, $zero, 10      # $v0 <-- function code for "exit"
                syscall                     # Syscall to exit

                # ***************************************************************************
                #               P R O J E C T   R E L A T E D   D A T A   S E C T I O N
                # ***************************************************************************
                .data                       # place variables, arrays, and constants, etc. in
                                              this area
```

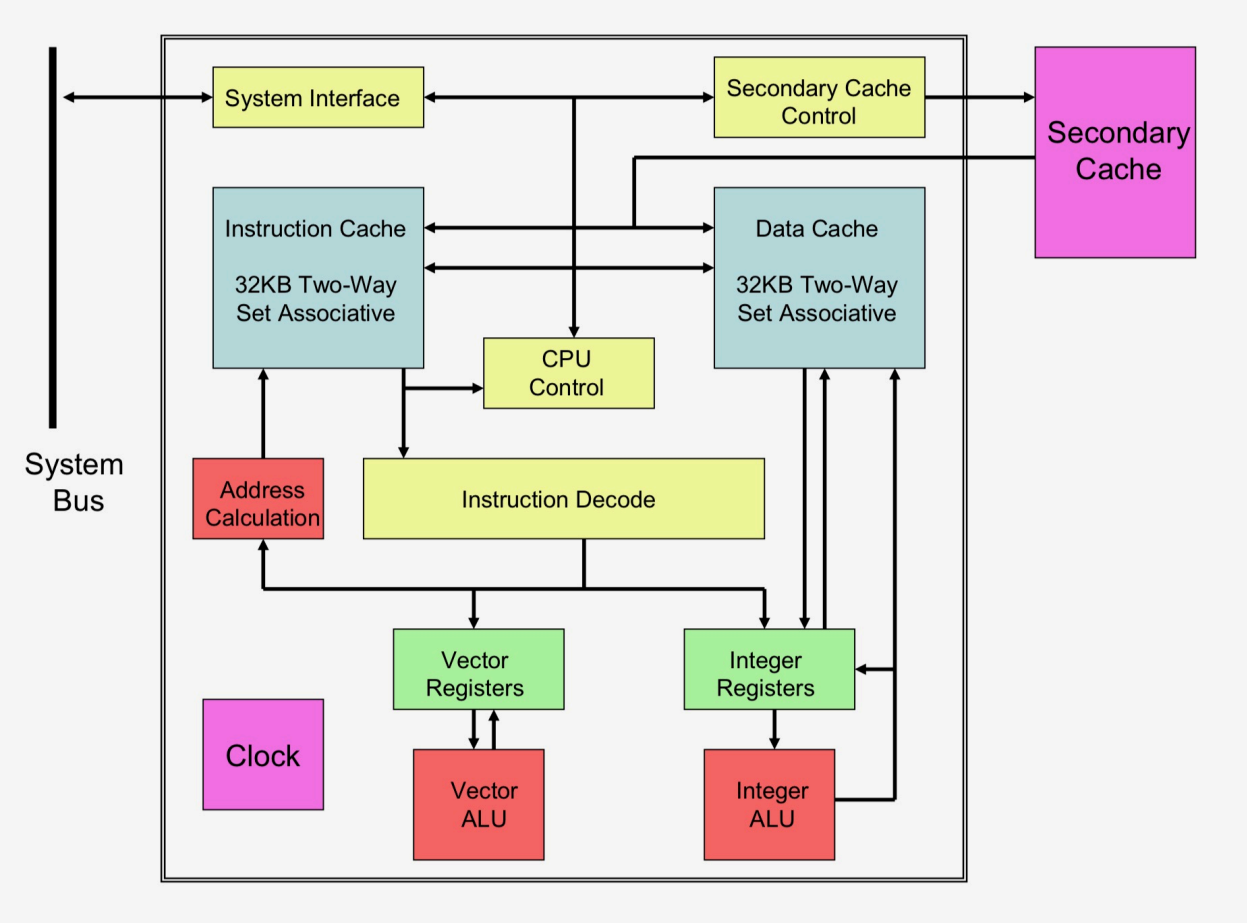
Before:

Name	Number	Value
$zero	0	0x00000000
$at	1	0x11220000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0xaabbccdd
$s1	17	0x11223344
$s2	18	0x00000001
$s3	19	0x00000004
$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		0x0040001c
hi		0x00000000
lo		0x00000000

After

Name	Number	Value
$zero	0	0x00000000
$at	1	0xff000000
$v0	2	0x0000000a
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0xaabbccdd
$s1	17	0x11223344
$s2	18	0x00000001
$s3	19	0x00000004
$s4	20	0xaa11ccdd
$s5	21	0xbb223344
$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		0x00400164
hi		0x00000000
lo		0x00000020

IV. Datapath Block Diagram



V. Additional Discussion/ Comments

Our Takeaways

These project has been a blessing in disguise. It was extremely challenging and difficult for us to implement these many different instructions and not only that but give a details and examples of to whom and how they should be useful. The coding part was the most challenging and mind stressing part. But once we got the gist of things, the implementation came a lot more quickly. The report itself was very tedious and extremely time consuming.

This was not just any ordinary school project. This is one of the biggest and most challenging ones we have had. Even though it took us so much time and effort to complete this project, we feel that we have learned so much about SIMD MIPS Instruction Set Architecture. What was really useful about this project is that not only did it help us understand the material but it gave us more experience of what it is like to work as a team. We learned that it is about communication, making deadlines, time management, and taking it one step at a time. In doing all these thing any project, no matter how big, will be that much easier to complete. So in conclusion, we are actually grateful for this assignment because even though it was hard work, it came with many learning experiences that are essential to have for our future careers.