

Project


Project for ECE352:


There are two options for your project. **By default, your project is designing a SIMD (vector) extension for a multicycle processor.**

For interested students, there is another option which is pipelining a multicycle processor.

If you want to work on the pipelining project you need to **send an email** to Prof. Moshovos and ask for his approval.

Again, by default, the project for all students is Vector Extension.

The file attached here provides the **starter code** for both projects and also describes what is needed for pipelining. [ECE352_Project.zip \(https://q.utoronto.ca/courses/238664/files/17254942?wrap=1\)](https://q.utoronto.ca/courses/238664/files/17254942?wrap=1) 
(https://q.utoronto.ca/courses/238664/files/17254942/download?download_frd=1)

This **video** helps you to get started with the project: https://utoronto.zoom.us/rec/play/ydQTj-wq-0F5AY_wYx7RCMc-RxFIYNo490ZDoG1q22exBKSXdlHy5duzlPuJs8n8jhVUJiQ8a-fimqMQ.-VF6m1JUUpJkr1yon  (https://utoronto.zoom.us/rec/play/ydQTj-wq-0F5AY_wYx7RCMc-RxFIYNo490ZDoG1q22exBKSXdlHy5duzlPuJs8n8jhVUJiQ8a-fimqMQ.-VF6m1JUUpJkr1yon)

Vector Extensions

Andreas Moshovos

As our project this year we will be extending our multicycle implementation with vector instructions. Our architecture as described implements *scalar* operations only, where a scalar, for our purposes is defined as an 8bit binary quantity.

Several processors today implement another class of instructions where the operands are vectors, where each vector is a small unidimensional array of scalars. For example, we can define vectors to be an array of 4 integers such as (10, 20, 3, 5). A vector instruction then operates on such vector inputs and produces vector outputs where operations are done element-wise. For example a vector add of (10, 20, 3, 5) with (20, 20, 1, 4) would produce (30, 40, 4, 9).

Why are such instructions useful? Because they allow us to perform multiple operations in parallel and thus produce more work per unit of time. This reduces the number of instructions needed and also allows us to completely overlap the element-wise operations.

Deliverables and Marking Scheme:

Lab of Nov. 22: Adding two instructions: "nop" and "stop" (step 1 in the previously provided handout--Pipeline) and the counter (step 2 in that handout). Use the following codes for the nop and stop instructions (not those in the previous handout). We don't implement step 3 or anything further from that handout.

During the lab, you will demo your design for TAs, and also you can ask any questions about the project.

This week's deliverables are **5%** of your project marks.

- **nop: 10000001**
- **stop: 00000001**

Lab of Nov. 29: Implementing the datapath of the vector unit in Verilog. You don't need to finish implementing the control unit by this week.

TAs will check your progress during the lab.

This week's deliverables are **30%** of your project marks.

Lab of Dec. 6: The processor with the vector unit should be ready for demo. You are responsible to test the processor with various test cases to make sure the design is working as expected.

Submit your files including the **Verilog code** of your completed design and **your test programs in assembly and binary** on Quercus before the lab.

This week's deliverables are **65%** of your project marks.

Let's make things specific.

Below is the original definition of our instruction architecture where we highlight the extensions (red).

Instruction Set Definition

We will use the following instruction set/behavior model for our examples. Assume a hypothetical architecture where there are just 256 memory locations. Our CPU has four general-purpose registers called K0 through K3 each capable of storing just 8 bits. The CPU also has a PC register which is also 8 bits. Upon resetting the CPU (e.g., power-up), the PC takes the value \$80. Our CPU does not support interrupts. In addition, there are two condition code bits Z = zero and N = negative.

Finally, our CPU has 4 4-element vector registers V0, V1, V2, and V3. Each vector register is 4B wide (32b) and is to be interpreted as a vector of 4 values where each value is a byte.

Our CPU is capable of executing the following 10+3 instructions:

LOAD R1 (R2) :

TMP = MEM [R2]

R1 = TMP

PC = PC + 1

STORE R1 (R2) :

MEM [R2] = R1

PC = PC + 1

ADD R1 R2

TMP = R1 + R2

R1 = TMP

IF (TMP == 0) ZERO = 1; ELSE ZERO = 0;

IF (TMP < 0) NEGATIVE = 1; ELSE NEGATIVE = 0;

PC = PC + 1

SUB R1 R2

```
TMP = R1 - R2
```

```
R1 = TMP
```

```
IF (TMP == 0) ZERO = 1; ELSE ZERO = 0;
```

```
IF (TMP < 0) NEGATIVE = 1; ELSE NEGATIVE = 0;
```

```
PC = PC + 1
```

NAND R1 R2

```
TMP = R1 bitwise NAND R2
```

```
R1 = TMP
```

```
IF (TMP == 0) ZERO = 1; ELSE ZERO = 0;
```

```
IF (TMP < 0) NEGATIVE = 1; ELSE NEGATIVE = 0;
```

```
PC = PC + 1
```

ORI IMM5

```
TMP = K1 bitwise OR IMM5, where IMM5 is a 5 bit constant
```

```
K1 = TMP
```

```
IF (TMP == 0) ZERO = 1; ELSE ZERO = 0;
```

```
IF (TMP < 0) NEGATIVE = 1; ELSE NEGATIVE = 0;
```

```
PC = PC + 1
```

SHIFT L/R R1 IMM2

```
IF (L) THEN TMP = R1 << IMM2
```

```
ELSE TMP = R1 >> IMM2
```

```
R1 = TMP
```

```
IF (TMP == 0) ZERO = 1; ELSE ZERO = 0;
```

```
IF (TMP < 0) NEGATIVE = 1; ELSE NEGATIVE = 0;
```

```
PC = PC + 1
```

Alternative definition (equivalent to the previous one):

IMM3 is a 3 bit immediate in the sign - magnitude representation, i.e., bit 2 = sign, value = bits 1 and 0

```

IF (IMM3 > 0) THEN  TMP = R1 << IMM3

                ELSE TMP = R1 >> (-IMM3)

R1 = TMP

IF (TMP == 0) ZERO = 1; ELSE ZERO = 0;

IF (TMP < 0) NEGATIVE = 1; ELSE NEGATIVE = 0;

PC = PC + 1

```

BZ IMM4

where IMM4 is a 2's complement 4-bit immediate

```

IF (ZERO == 1) PC = PC + 1 + (SIGN-EXTEND8(IMM4))

ELSE PC = PC + 1

```

BNZ IMM4

IMM4 as in BZ

```

IF (ZERO == 0) PC = PC + 1 + (SIGN-EXTEND8(IMM4))

ELSE PC = PC + 1

```

BPZ IMM4

IMM4 as in BZ

```

IF (NEGATIVE == 0) PC = PC + 1 + (SIGN-EXTEND8(IMM4))

ELSE PC = PC + 1

```

VLOAD X1 (R2):

```

TMP0 = MEM[R2];

TMP1 = MEM[(R2+1) MOD 256];

TMP2 = MEM[(R2+2) MOD 256];

TMP3 = MEM[(R2+3) MOD 256];

```

$X1 = (TMP0, TMP1, TMP2, TMP3)$ where $X1$ a vector register

$PC = PC + 1$

VSTORE X1 (R2) :

$MEM[R2] = TMP0;$

$MEM[(R2+1) \text{ MOD } 256] = TMP1;$

$MEM[(R2+2) \text{ MOD } 256] = TMP2;$

$MEM[(R2+3) \text{ MOD } 256] = TMP3;$

$PC = PC + 1$

VADD X1 X2

$X1 = X1 + X2$ (element wise addition of vector registers $X1$ and $X2$)

For example, the following sequence of instructions loads two 4 element vectors into $V0$ and $V1$ adds them and writes the result back to memory

ORI 0x10 # K1 = 0x10

VLOAD V0 (K1) # vector load starting from address 0x10 into V0

ORI 0x14

VLOAD V1 (K1) # vector load starting from address 0x14 into V1

VADD V0 V1 # vector add V0 and V1 result into V0

ORI 0x10

VSTORE V0 (K1) # write V0 starting at address 0x10

If for example, memory starting at address 0x10 had the following contents:

Address\Offset	+0	+1	+2	+3

0x10	5	6	7	8
0x14	1	2	3	4

After the preceding instruction executes memory will be as follows:

Address\Offset	+0	+1	+2	+3
0x10	6	8	10	12
0x14	1	2	3	4

And registers V0 and V1 will contain respectively (6,8,10,12) and (1,2,3,4) or expressed as 32b quantities 0x06080a0c and 0x01020304.

Each instruction is encoded in one byte as follows:

LOAD

7	6	5	4	3	2	1	0
R1	R1	R2	R2	0	0	0	0

STORE

7	6	5	4	3	2	1	0
R1	R1	R2	R2	0	0	1	0

ADD

7	6	5	4	3	2	1	0
R1	R1	R2	R2	0	1	0	0

SUB

7	6	5	4	3	2	1	0
R1	R1	R2	R2	0	1	1	0

NAND

7	6	5	4	3	2	1	0
R1	R1	R2	R2	1	0	0	0

VLOAD

7	6	5	4	3	2	1	0
X1	X1	R2	R2	1	0	1	0

VSTORE

7	6	5	4	3	2	1	0
X1	X1	R2	R2	1	1	0	0

VADD

7	6	5	4	3	2	1	0
X1	X1	X2	X2	1	1	1	0

ORI

7	6	5	4	3	2	1	0
IMM5_4	IMM5_3	IMM5_2	IMM5_1	IMM5_0	1	1	1

SHIFT

7	6	5	4	3	2	1	0
R1	R1	L!/R	IMM2_1	IMM2_0	0	1	1

BZ

7	6	5	4	3	2	1	0
IMM4_3	IMM4_2	IMM4_1	IMM4_0	0	1	0	1

BNZ

7	6	5	4	3	2	1	0
IMM4_3	IMM4_2	IMM4_1	IMM4_0	1	0	0	1

BPZ

7	6	5	4	3	2	1	0
IMM4_3	IMM4_2	IMM4_1	IMM4_0	1	1	0	1

Below we show the beginnings of a modified multicycle datapath that supports the vector extensions. Your responsibilities include implementing a correct datapath and control. The datapath shown is meant as an aid. **You are responsible for figuring out what a correct datapath and its associated control should be.**

The extensions include:

A Vector Register File (VRF) with 4 4x8b registers V0 through V1. The register file has two read ports. The read ports are (vreg1, vdata1) and (vreg2, vdata2) where vreg1 and vreg2 register numbers (2b) and vdata1 and vdata2 are 4x8b values. The read ports are combinatorial, no clock is needed. There is a single write port (vregw, vdataw, VRFWrite) where vregw a 2b register number, vdataw a 4x8b data values, and VRFwrite the write enable signal. This is edge triggered.

Two temporary vector registers X1 and X2 each connected directly to a read data port. The signals X1Load and X2Load are write enables and both registers are edge triggered.

four 8b adders connected to the corresponding pairs of 8b elements of registers X0 and X1.

Four 8b temporary registers T0 through T3 edged triggered, each with a write enable T0Ld through T3Ld. The T registers can accept inputs from the four adders (each T register is connected to one adder) or from memory (all T registers are connected to the same 8b output from memory).

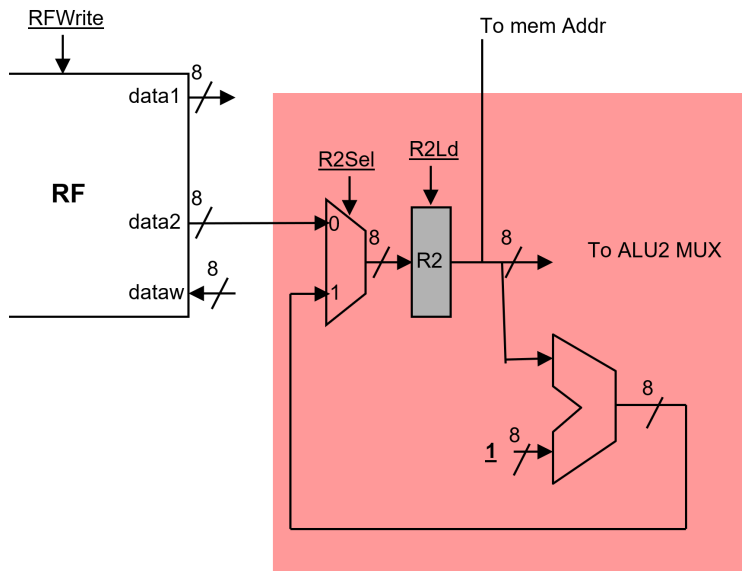
A multiplexer at the input of memory which can be used to select either the R1 (as in the original datapath) or any of the four bytes within register X1.

Here's how this datapath can be used to implement the three new instructions:



We follow the same steps as a regular LOAD A (B) instruction. However, after reading register B into R2, we start then to read one byte a time from memory loading them into the T registers. Once all T0 through T3 are loaded from memory we write them to the VRF in one step.

For this instruction, you will have to modify the datapath around R2 to introduce the ability to increment R2 by 1. A multiplexer and an adder (with +1) should be sufficient. That is R2 should maintain the ability to be loaded from the RF and we should also add another option to load R2 with its current value plus 1. This is show below.



VSTORE A (B)

Where A one of the V vector registers and B one of the K registers.

First we read the A vector register into X1 and the B register. Then over four cycles, we can write each of the 8b values in X1 into memory while also incrementing B by 1 each time (using the extension explained above)

VADD A B

Where A and B are both V vector registers.

We first read A and B into X1 and X2 respectively. Then perform the element-wise addition through the four adders writing the results into the T registers. Then write the T register using a single cycle into the VRF.

EXTRA CREDIT

This part is optional. We will introduce a set of new instructions:

VMUL A B where A and B vector registers

MOVI A Imm8 where A a K register and Imm8 an 8b immediate.

We will also increase the number of vector registers to 16. So now we have V0 up to V15.

To fit all these instructions will be changing the encoding of VLOAD, VSTORE, and VADD too. All these instructions (MOVI, VMUL, VADD, VSTORE, VLOAD) will now use 2 bytes.

The encoding will be as follows:

MOVI A Imm8

1st byte

7	6	5	4	3	2	1	0
A	A	0	0	1	1	1	0

2nd byte

7	6	5	4	3	2	1	0
Imm8							

VLOAD A (B)

1st byte

7	6	5	4	3	2	1	0
---	---	---	---	---	---	---	---

0	0	B	B	1	0	1	0
---	---	---	---	---	---	---	---

2nd byte

7	6	5	4	3	2	1	0
A	A	A	A	0	0	0	0

VSTORE A (B)1st byte

7	6	5	4	3	2	1	0
0	1	B	B	1	0	1	0

2nd byte

7	6	5	4	3	2	1	0
A	A	A	A	0	0	0	0

VADD A B1st byte

7	6	5	4	3	2	1	0
1	0	0	0	1	0	1	0

2nd byte

7	6	5	4	3	2	1	0
A	A	A	A	B	B	B	B

VMUL A B1st byte

7	6	5	4	3	2	1	0
1	1	0	0	1	0	1	0

2nd byte

7	6	5	4	3	2	1	0
A	A	A	A	B	B	B	B

Modify the datapath and control to implement these instructions.