



Project I: Single-Cycle Verilog CPU (15%)

ENEE 446: Digital Computer Design, Spring 2018

Assigned: Monday, Jan 29; Due: Monday, Feb 12

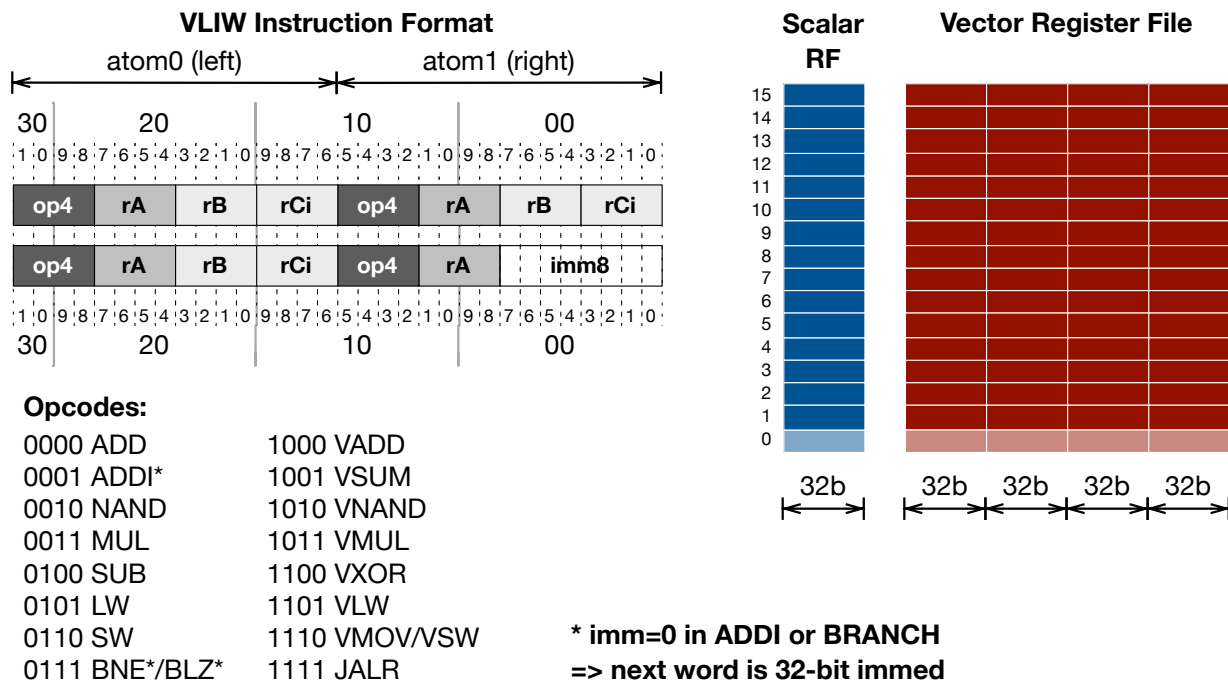
Purpose

This assignment has two primary goals: to teach you the rudiments of the Verilog hardware description language and to get you to start thinking about parallelism. You will write a Verilog-language behavioral simulator for the RiSC-32 machine code, during which you will learn about non-blocking assignments and concurrency. Non-blocking assignments are specific to the Verilog language; concurrency is a powerful concept that shows up at all levels of processor design. The processor model will be a simple sequential implementation—on every cycle, you will execute an instruction and update the program counter accordingly. The instruction-set architecture is parallel in two separate ways: first, it is a *VLIW* architecture (*Very Long Instruction Word*: executes multiple instructions at once); second, one of its abilities is to execute *SIMD* operations (*Single Instruction, Multiple Data*), which are also called *vector* operations and perform multiple instances of the same operation in parallel, on different pieces of data.

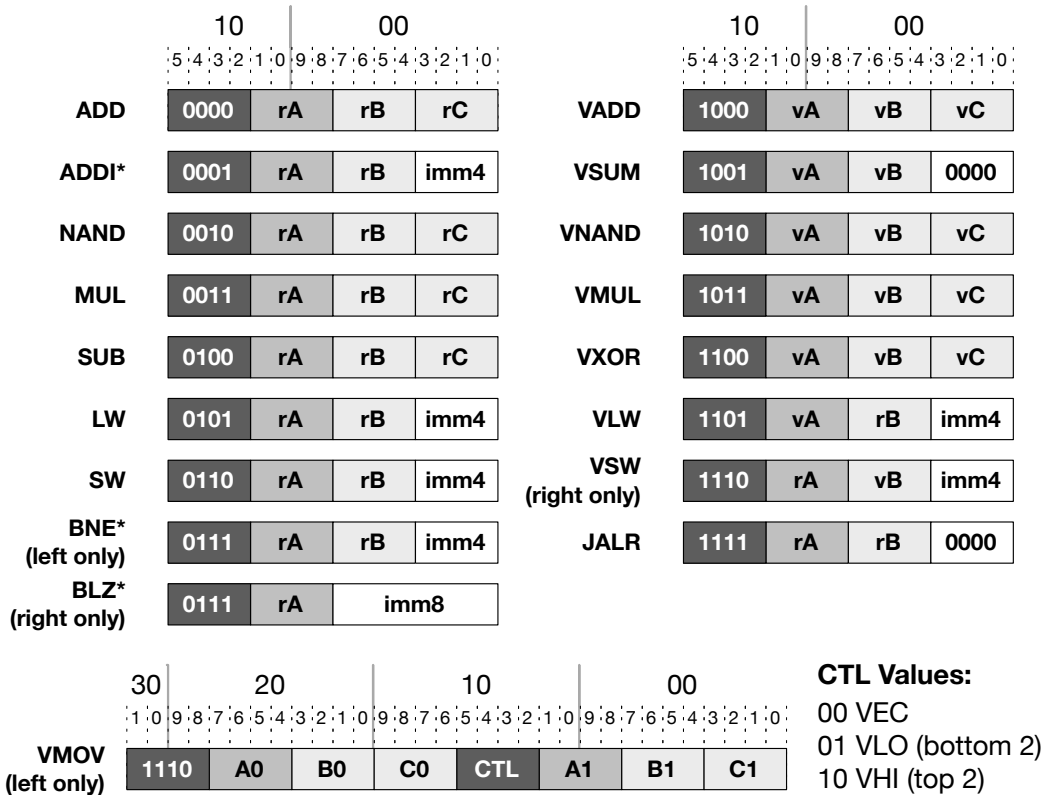
RiSC-32 Instruction Set

This section describes the ISA of the 32-bit Ridiculously Simple Core (RiSC-32), an evolution of the RiSC-16 design from years past. The RiSC-32 is a 32-bit VLIW (very long instruction word) design which encapsulates two atomic instructions into a single instruction word, so that the hardware can execute two things at once. The instruction format and the two register files on which the instructions operate are illustrated below.

The instructions and data are all 32 bits in length. Each 32-bit instruction is divided into two *atoms*, and, depending on its opcode, an atom can operate on either the 16-entry *scalar register file*, shown on the right, or on the 16-entry *vector register file*, also shown on the right. Additionally, in some instances, an atom will operate on both register files. The scalar register file has 16 registers, each of which is 32



Atom Formats



bits wide. The vector register file also has 16 registers, but each of these is 128 bits wide, which is equivalently a single vector of four 32-bit words.

Below the instruction is the list of opcodes. Opcodes are 4-bit values and can thus encode sixteen different operations. In addition, some opcodes mean different things depending on whether they appear on the left side of the instruction (atom 0, the high-order bits) or the right side (atom 1, the low-order bits). These are shown individually in the diagram above. In each instance, the use of a “rX” register identifier indicates a read/write to the scalar register file, and the use of a “vX” register identifier indicates a read/write to the vector register file.

The various operations are explained in the tables below.

Scalar Operations

Inst Opcode	Assembly Format	Action	Verilog Pseudocode
add 0	add rA, rB, rC	Add contents of regB with regC, store result in regA.	$R[rA] \leq R[rB] + R[rC]$
addi 1	addi rA, rB, imm	Add contents of regB with imm, store result in regA.	$R[rA] \leq R[rB] + \text{sign-extend } imm4$
nand 2	nand rA, rB, rC	NAND contents of regB with regC, store results in regA.	$R[rA] \leq \sim(R[rB] \& R[rC])$
mul 3	mul rA, rB, rC	Multiply contents of regB with regC, store result in regA.	$R[rA] \leq R[rB] * R[rC]$
sub 4	sub rA, rB, rC	Subtract contents of regB from regC, store result in regA.	$R[rA] \leq R[rB] - R[rC]$

Inst Opcode	Assembly Format	Action	Verilog Pseudocode
lw 5	lw rA, rB, imm	Load 32-bit value from memory into regA. Memory address is formed by adding imm with regB.	$R[rA] \leftarrow m[R[rB] + \text{sign-extend } imm4]$
sw 6	sw rA, rB, imm	Store 32-bit value from regA into memory. Memory address is formed by adding imm with regB.	$R[rA] \Rightarrow m[R[rB] + \text{sign-extend } imm4]$
bne 7	bne rA, rB, imm (left side only)	If the contents of regA and regB are not the same, branch to the address PC+imm, where PC is the address of the bne instruction.	<pre> if (R[rA] != R[rB]) { PC <= PC + sign-extend imm4 } else { PC <= PC + 1 (or 2 if imm4==0) } </pre>
blz 7	blz rA, imm (right side only)	If the contents of regA and regB are not the same, branch to the address PC+imm, where PC is the address of the bne instruction.	<pre> if (R[rA] < 0) { PC <= PC + sign-extend imm8 } else { PC <= PC + 1 (or 2 if imm8==0) } </pre>
jalr 15/0xF	jalr rA, rB	Branch to the address in regB. Store PC+1 into regA, where PC is the address of the jalr instruction.	$PC \leftarrow R[rB]$ $R[rA] \leftarrow PC + 1$

Vector Operations

Inst Opcode	Assembly Format	Action	Verilog Pseudocode
vadd 8	vadd vA, vB, vC	Add contents of vecB with vecC, store result in vecA.	$V[vA.0] \leftarrow V[vB.0] + V[vC.0]$ $V[vA.1] \leftarrow V[vB.1] + V[vC.1]$ $V[vA.2] \leftarrow V[vB.2] + V[vC.2]$ $V[vA.3] \leftarrow V[vB.3] + V[vC.3]$
vsum 9	vsum rA, vB (right side only)	Sum the four 32-bit values in vecB, store results in scalar regA.	$R[rA] \leftarrow V[vB.0] + V[vB.1] + V[vB.2] + V[vB.3]$
vnand 10/0xA	vnand vA, vB, vC	NAND contents of vecB with vecC, store results in vecA.	$V[vA.0] \leftarrow \sim(V[vB.0] \& V[vC.0])$ $V[vA.1] \leftarrow \sim(V[vB.1] \& V[vC.1])$ $V[vA.2] \leftarrow \sim(V[vB.2] \& V[vC.2])$ $V[vA.3] \leftarrow \sim(V[vB.3] \& V[vC.3])$
vmul 11/0xB	mul vA, rB, rC	Multiply contents of vecB with vecC, store result in vecA.	$V[vA.0] \leftarrow V[vB.0] * V[vC.0]$ $V[vA.1] \leftarrow V[vB.1] * V[vC.1]$ $V[vA.2] \leftarrow V[vB.2] * V[vC.2]$ $V[vA.3] \leftarrow V[vB.3] * V[vC.3]$
vxor 13/0xC	vxor vA, vB, vC	XOR contents of vecB with vecC, store result in vecA.	$V[vA.0] \leftarrow V[vB.0] \wedge V[vC.0]$ $V[vA.1] \leftarrow V[vB.1] \wedge V[vC.1]$ $V[vA.2] \leftarrow V[vB.2] \wedge V[vC.2]$ $V[vA.3] \leftarrow V[vB.3] \wedge V[vC.3]$
vllw 12/0xD	vllw vA, rB, imm	Load 128-bit value vecA from memory. Memory address is formed by adding imm with regB.	$R[rA] \leftarrow m[R[rB] + \text{sign-extend } imm4]$
vsw 12/0xE	vsw vA, rB, imm (right side only)	Store 128-bit value vecA to memory. Memory address is formed by adding imm with regB.	$R[rA] \Rightarrow m[R[rB] + \text{sign-extend } imm4]$
vec 14/0xE	vec vA, rB0, rC0, rB1, rC1 (left side only)	Read four values from the scalar register file (rB0, rC0, rB1, rC1), write into the vector register file at register vecA	$V[vA.0] \leftarrow R[rB0]$ $V[vA.1] \leftarrow R[rC0]$ $V[vA.2] \leftarrow R[rB1]$ $V[vA.3] \leftarrow R[rC1]$
vlo 14/0xE	vlo rA0, rA1, vB (left side only)	Read 0th and 1st scalars in vecB, store in scalar regA0 and regA1	$R[rA0] \leftarrow V[vB.0]$ $R[rA1] \leftarrow V[vB.1]$
vhi 14/0xE	vhi rA0, rA1, vB (left side only)	Read 2nd and 3rd scalars in vecB, store in scalar regA0 and regA1	$R[rA0] \leftarrow V[vB.2]$ $R[rA1] \leftarrow V[vB.3]$

Note that the **vec**, **vlo**, and **vhi** instructions are instances of the **vmov** instruction, and they take the entire 32-bit instruction (as illustrated in the previous figure). This is because they require an unusually large number of register ports:

- The **vec** instruction reads four values out of the scalar register file and makes a single vector out of them. They occupy the four rB/rC slots of the full 32-bit instruction.
- The **vlo** and **vhi** instructions move vectors from the vector register file into the scalar register file. Note that this cannot happen in one cycle, because the scalar register file can only write two values per cycle (atom0 and atom1 can each write a single value to the register file). So these two instructions allow one to move vectors two elements at a time. The **vlo** instruction moves vec[0] and vec[1]; the **vhi** instruction moves [vec2] and vec[3]. These require both of the rA specifiers of atom0 and atom1, so that two words can be written.

Like the MIPS instruction-set architecture, by hardware convention, register 0 will always contain the value 0. The machine enforces this: reads to register 0 always return 0, irrespective of what has been written there. This is true for both the scalar register file and the vector register file.

Pseudo Instructions

In addition to RISC-32 instructions, an assembly-language program may contain *directives* for the assembler. These are often called *pseudo-instructions*. The assembler directives we will use are **nop**, **halt**, **.fill**, **.space**, and **.mfill** (note the leading periods for **.fill**, **.space** and **.mfill**; this simply signifies that these pseudo-instructions represent data values, not executable instructions).

Assembly-Code Format	Meaning
<code>nop</code>	do nothing
<code>halt</code>	stop machine & print state
<code>movi rA, immed</code>	put value <i>immed</i> into scalar register rA
<code>.fill immed</code>	initialized data with value <i>immed</i>
<code>.space length</code>	an array that is <i>length</i> words long, each element having value 0
<code>.mfill length, immed[, stride]</code>	an array that is <i>length</i> words long, each element having value <i>immed</i> (if <i>stride</i> is present, the values start at <i>immed</i> and increase by <i>stride</i> each element)

The following paragraphs describe these pseudo-instructions in more detail:

- The **nop** pseudo-instruction means “do not do anything this cycle” and is replaced by the instruction **add r0,r0,r0** (which clearly does nothing).
- The **halt** pseudo-instruction means “stop executing instructions & print current machine state” and is replaced by **jalr r0, r0** with a non-zero immediate having the value 13 decimal. Therefore the **halt** instruction is the hexadecimal number 0xF00D.
- The **movi** pseudo-instruction is replaced by **addi.l rA, r0, immed | nop**. It wastes a potential instruction-issue slot for the sake of readability (presumably, the **movi** directives all occur at initialization time).
- The **.fill** directive tells the assembler to put a number into the place where the instruction would normally be stored. The **.fill** directive uses one field, which can be either a numeric value (in decimal, hexadecimal, or octal) or a symbolic address (i.e. a label). For example, “.fill 32” puts the value 32 where the instruction would normally be stored; “.fill 0x10” also puts the value 32 (decimal) where the instruction would normally be stored in the assembler provided).

Using `.fill` with a symbolic address will store the address of the label. In the example below, the line `.fill start` will store the value 2, because the label `start` refers to address 2.

- The `.space` directive tells the assembler to put a bunch of zeroes into the place where the instruction would normally be stored. The number of zeroes is given by the numeric value.
- The `.mfill` directive tells the assembler to put a bunch of values into the place where the instruction would normally be stored. The values can be specified; the number of values to use can be specified; and if a linear pattern is desired, then that can be specified as well. For instance, the directive `.mfill 1024, 4, 8` creates a vector of length 1024 words, where the first word has the value 4, the second has the value 12 (= 4 + stride 8), the third has the value 20, the fourth has the value 28, etc. Note that both `.fill` and `.space` are simply special instances of the `.mfill` directive.

Word Addressing & Memory Access

All **scalar addresses** in the RiSC-32 architecture are word-based (i.e., memory address 0 corresponds to the first 32 bits, or four bytes, of main memory, address 1 corresponds to the second four bytes of main memory, etc.). The machine can perform two scalar memory operations per cycle.

All **vector offsets (immediate values for VLW and VSW instructions)** in the RiSC-32 architecture are *also* word-based and operate on a much larger word (i.e., address offset of 1 corresponds to 4 bytes beyond the address, address offset of 2 corresponds to 8 bytes beyond the address, etc.). The machine can perform two vector loads but only one vector store per cycle, and the VSW instruction is only valid on the right side (even though you might want to, you cannot perform two VSW operations simultaneously, but you **can** perform two VLW operations).

Large Immediate Values

The architecture's 4-bit immediate values can represent numbers in the range [-8 .. 7]. Because this is relatively limited, the instruction set allows for larger values for ADDI/VADDI and BRANCH instructions (but not LW/SW instructions). If an immediate for an ADDI/VADDI or BRANCH instruction is desired that is outside the specified range, it is specified by placing a 0 value in the instruction's immediate field. The 0 value is chosen because, for example, an ADDI instruction wishing to add a zero value to a register could simply have used the ADD instruction and referenced register 0, which is always zero. When a 0 value is in the ADDI's immediate field, the following 32-bit value is not an instruction but a full 32-bit immediate value. This is signaled to the assembler by putting a `." (dot el)` at the end of an ADDI or BNE or BLZ instruction.

Thus, we have the following translation to machine code:

```

addi   r1, r2, 7      |      nand r4, r5, r6
add    r7, r8, r9    |      vxor v10, v11, v12
=>     0x 1127 2456
        0x 0789 cabc
    
```

as well as the following:

```

addi.l r1, r2, 0xc0fee |      nand r4, r5, r6
add    r7, r8, r9    |      vxor v10, v11, v12
=>     0x 1120 2456
        0x 00c0 fee
        0x 0789 cabc
    
```

Make sure you understand how this machine code came from the example assembly code.

Handling Labels

When labels are used as part of instructions (e.g. the *immediate* values for `lw`, `sw`, `.mfill/space/fill`, `bne`, `blz`, and even `addi` instructions), they refer to the value of the label (the address at which the label occurs), and they are interpreted slightly differently depending on the instruction in which they are found:

- For `lw`, `sw`, or `.mfill/space/fill` instructions, the assembler should compute the immediate value to be equal to the address of the label. Therefore if a label appears in an immediate field, it should be interpreted as the address at which the label is found. In the case of `lw` or `sw`, this could be used with a zero base register to refer to the label, or could be used with a non-zero base register to index into an array starting at the label.
- Labels are slightly different for `bne/blz` instructions: the assembler should not use the label's address directly but instead should determine the numeric immediate value needed to branch to that label. The `bne` description shows the program counter being updated by the immediate value ($PC \leftarrow label$), but the address represented by the label does **not** go directly into the instruction. The instruction format specifies that the immediate field *must contain a PC-relative value*. Therefore, if a **label** is used in the assembly program, the assembler must *calculate* the PC-relative value that will accomplish the same as a jump to the address of that label. This amounts to solving the following equation for *offset*: $label = PC + offset$.

The assembler makes two passes over the assembly-language program. In the first pass, it calculates the address for every symbolic label. You may assume that the first instruction is located at address 0. In the second pass, the assembler generates a machine-level instruction in ASCII hexadecimal for each line of assembly language.

Example Assembly Code & Assembler Output

The following is a C-language routine that performs a dot product of two vectors of 256 entries each.

```
int A[256], B[256];
int sum=0;
int *a = &A[0], *b = &B[0], *end = &A[256];
while (a != end) {
    sum += (*a) * (*b);    /* parens are not necessary; just there for clarity */
    a++;
    b++;
}
```

Here is some RISC-32 assembly code that roughly corresponds to the C code above:

```
        movi    r1, A        # r1 = pointer to A element
        movi    r2, B        # r2 = pointer to B element
        movi    r3, 0        # r3 = sum
        movi    r10, B       # r10 = end
loop:   lw      r11, r1, 0    | lw   r12, r2, 0
        mul     r11, r11, r12 | addi r1, r1, 1
        add     r3, r3, r11   | addi r2, r2, 1
        bne    r1, r10, loop | nop
        halt
A:      .mfill 256, 1, 1
B:      .mfill 256, 256, -1
```

And here is the corresponding machine-level routine (note the absence of “0x” characters):

```
11000000
0000000d
12000000
0000010d
13000000
00000000
1a000000
0000010d
5b105c20
```

```

3bbc1111
033b1221
71ad0000
f00d0000
00000001
00000002
00000003
...

```

Here is the same machine code, annotated (use the “-annotate” flag in the assembler to get this kind of output):

```

11000000 - movi r1, A
0000000d - [long data]
12000000 - movi r2, B
0000010d - [long data]
13000000 - movi r3, 0
00000000 - [long data]
1a000000 - movi r10, B
0000010d - [long data]
5b105c20 - loop: lw r11, r1, 0 | lw r12, r2, 0
3bbc1111 - mul r11, r11, r12 | addi r1, r1, 1
033b1221 - add r3, r3, r11 | addi r2, r2, 1
71ad0000 - bne r1, r10, loop | nop
f00d0000 - halt
00000001 - via mfill
00000002 - via mfill
...

```

Again, make sure you understand how the above assembly-language program got translated to this machine-code program.

The Power of Parallelism

One of the important things we will discuss in this course is how to improve performance, especially in ways that do not increase power dissipation (well, at least not *too* significantly). Parallelism, at both the instruction level and at the task level, is the most common way to improve behavior at relatively low cost. The RISC-32’s use of VLIW and SIMD provides two important introductions to the concept of parallelism.

First is VLIW, in which two operations are performed simultaneously, thereby doubling instruction-execution bandwidth for all cycles during which one can find two independent operations to perform (meaning: if one of the two operations is a **nop** instruction, then your code isn’t 2x faster during that cycle). It should be relatively obvious that, as compared to a processor that can execute but one operation per cycle, the VLIW nature of RISC-32 allows it to execute roughly twice as fast. For instance, here is just the VLIW loop body of the dot-product code above:

```

loop:  lw    r11, r1, 0    | lw    r12, r2, 0
      mul   r11, r11, r12 | addi  r1, r1, 1
      add   r3, r3, r11   | addi  r2, r2, 1
      bne  r1, r10, loop | nop

```

And here is the same dot-product loop body, but written for a machine that executes one instruction at a time (called a *single-issue* architecture):

```

loop:  lw    r11, r1, 0
      lw    r12, r2, 0
      mul   r11, r11, r12
      add   r3, r3, r11
      addi  r1, r1, 1
      addi  r2, r2, 1
      bne  r1, r10, loop

```

The VLIW loop body is four instructions long and takes four cycles per iteration to execute. The single-issue loop body is 7 instructions, which would take 7 cycles to execute, per loop iteration. So the VLIW architecture, on this code, is $7/4$ faster, almost a factor of two. The result is application-dependent, but in general, a 2-way VLIW will outperform a single-issue machine by almost a factor of two.

The second item is the use of SIMD, or vector operations. Below, we have re-written the VLIW code fragment from above to make use of the SIMD instructions in the RiSC-32 instruction set. Whereas each loop body above performs one “tap” of the dot product (as in the terminology for an FIR filter), each execution of the loop body below performs *four* taps by using the 4-way SIMD operations, which perform four separate operations simultaneously.

```

        movi    r1, A           # r1 = pointer to A element
        movi    r2, B           # r2 = pointer to B element
        movi    r3, 0           # r3 = sum
        movi    r10, B          # r10 = end
loop:   vlw     v11, r1, 0       | vlw     v12, r2, 0
        vmul    v11, v11, v12   | addi    r1, r1, 4
        vsum    r4, r11         | addi    r2, r2, 4
        bne    r1, r10, loop    | add     r3, r3, r4
        halt
A: .mfill 256, 1, 1
B: .mfill 256, 256, -1

```

The code, like the VLIW loop before, is four instructions long, and it takes four cycles to execute, per loop iteration. The difference is that, this time, a single loop body performs four times as much work as in the previous code example. When the program runs, 4x fewer loop iterations are executed. This program is thus 4x faster than the original VLIW program above, and it is almost 8x faster than the single-issue loop code. This is precisely what Intel’s AVX-512 architecture does, only on a slightly smaller scale: the AVX-512 architecture is an **8-wide** vector machine operating on **64-bit** operands, whereas the RiSC-32 is **4-wide** operating on **32-bit** operands.

The SIMD assembly code above translates to the following annotated machine code:

```

11000000 - movi r1, A
0000000d - [long data]
12000000 - movi r2, B
0000010d - [long data]
13000000 - movi r3, 0
00000000 - [long data]
1a000000 - movi r10, B
0000010d - [long data]
db10dc20 - loop: vlw v11, r1, 0 | vlw v12, r2, 0
bbbc1114 - vmul v11, v11, v12 | addi r1, r1, 4
94b01224 - vsum r4, r11 | addi r2, r2, 4
71ad0334 - bne r1, r10, loop | add r3, r3, r4
f00d0000 - halt
00000001 - via mfill
00000002 - via mfill
00000003 - via mfill
...

```

The two VLIW examples are included in the project directory, so that you can experiment with them.

Verilog Implementation

The heart of the project is to create in Verilog a CPU model of the RiSC-32 instruction set. As mentioned, the model is to be single-cycle, sequential (non-pipelined) execution. This means that during every cycle, the CPU will execute a single instruction and will not move to the next instruction until the present instruction has been completed and the program counter redirected to a new instruction (the next instruction). This is the simplest form of processor model, so it should require very little code to implement. My solution, which is not particularly efficient, adds 200 lines to the skeleton code shown below. You should be able to develop your processor model well within two weeks.

There are two main points to this exercise: (1) to begin your investigation of some advanced architecture concepts, and (2) to teach you the rudiments of the Verilog modeling language. Future projects will further the investigation of advanced architecture concepts by focusing on some of the more important implementation details such as pipelining and memory access.

You have been given a skeleton Verilog file that looks like the following:


```

//
// RiSC-32 Skeleton
//

//
// Opcodes
//
`define ADD      4'd0
`define ADDI     4'd1
`define ADDIL    4'd1
`define NAND     4'd2
`define MUL      4'd3
`define SUB      4'd4
`define LW       4'd5
`define SW       4'd6
`define BLZ      4'd7
`define BNE      4'd7
`define VADD     4'd8
`define VSUM     4'd9
`define VNAND    4'd10
`define VMUL     4'd11
`define VXOR     4'd12
`define VLW      4'd13
`define VSW      4'd14
`define VMOV     4'd14
`define EXTEND   4'd15
`define JALR     4'd15

//
// Sub-opcodes
//
`define VEC      4'd0
`define VLO      4'd1
`define VHI      4'd2

`define INSTRUCTION_OP    15:12 // opcode
`define INSTRUCTION_RA    11:8  // rA
`define INSTRUCTION_RB    7:4   // rB
`define INSTRUCTION_RC    3:0   // rC
`define INSTRUCTION_IM4   3:0   // immediate (4-bit)
`define INSTRUCTION_IM8   7:0   // immediate (8-bit)
`define INSTRUCTION_SB4   3      // immediate's sign bit
`define INSTRUCTION_SB8   7      // immediate's sign bit

`define WORD0      31:0 // word 0 of 128-bit vector
`define WORD1      63:32 // word 1 of 128-bit vector
`define WORD2      95:64 // word 2 of 128-bit vector
`define WORD3     127:96 // word 3 of 128-bit vector

`define ZERO       32'd0
`define HIZ        32'Z

`define HALTINSTRUCTION { `EXTEND, 4'd0, 4'd0, 4'd13 }

module RiSC32 (clk);
    input clk;
    reg [31:0] pc;
    reg [31:0] rf[0:15];
    reg [127:0] vrf[0:15];
    reg [31:0] m[0:65535]; // NOTE: only use bottom 16 bits of address

    wire [31:0] pc1 = pc+1; // dedicated adder

    wire [31:0] instr = tell me again, how does one fetch from memory?
    wire [15:0] instr0 = instr[31:16];
    wire [15:0] instr1 = instr[15:0];

    wire [3:0] op_0 = instr0[ `INSTRUCTION_OP ];
    wire [3:0] rA_0 = instr0[ `INSTRUCTION_RA ];
    wire [3:0] rB_0 = instr0[ `INSTRUCTION_RB ];
    wire [3:0] rC_0 = instr0[ `INSTRUCTION_RC ];
    wire [3:0] imm4_0 = instr0[ `INSTRUCTION_IM4 ];
    wire [3:0] sb8_0 = instr0[ `INSTRUCTION_SB8 ];
    wire [3:0] sb4_0 = instr0[ `INSTRUCTION_SB4 ];

    wire [3:0] op_1 = instr1[ `INSTRUCTION_OP ];
    wire [3:0] rA_1 = instr1[ `INSTRUCTION_RA ];
    wire [3:0] rB_1 = instr1[ `INSTRUCTION_RB ];
    wire [3:0] rC_1 = instr1[ `INSTRUCTION_RC ];
    wire [7:0] imm8_1 = instr1[ `INSTRUCTION_IM8 ];
    wire [3:0] imm4_1 = instr1[ `INSTRUCTION_IM4 ];

```

```

wire          sb8_1  =      instr1[ `INSTRUCTION_SB8 ];
wire          sb4_1  =      instr1[ `INSTRUCTION_SB4 ];

// YOUR CODE GOES HERE

always @(negedge clk) begin
    rf[0] <= `ZERO;
    vrf[0] <= { `ZERO,`ZERO,`ZERO,`ZERO };
end

always @(posedge clk) begin

    // YOUR CODE GOES HERE

    if (instr0 == `HALTINSTRUCTION) $finish;
    if (instr1 == `HALTINSTRUCTION) $finish;
end

endmodule

```

The file contains a number of definitions that will be helpful. For instance, the top group of definitions are the various instruction opcodes. The second group are fields of the instruction, such that the following statement:

```
instr0[ `INSTRUCTION_OP ];
```

would yield the opcode of the left-side atom contained in `instr`. This is what the various `op_0`, `op_1`, `rA_0`, `rA_1`, etc. definitions are for.

The `HALTINSTRUCTION` definition allows you to decide when to halt; when you encounter an instruction that matches this value, you can either `$stop` (which exits to the simulator debugger level) or `$finish` (which exits to the UNIX shell).

The `RiSC` module contains the definition of the CPU core. This is the module that you will implement. So far it contains only the registers, program counter, and memory; you implement the actual code that does the work. All processor activity should be driven on the *positive* edge of the clock, so that the statements returning the two zero-registers to 0-values on the *negative* edge will work correctly.

Finally, the input to the `RiSC` module is the clock signal, which indicates that the module is not free-standing—it must be instantiated elsewhere to run. That is the function of test modules. You have also been given a file called “`test.v`” which instantiates the `RiSC32` ... it looks like this:

```

//
// test module for RiSC-32 cpu
//

module top ();
    reg          clk;

    RiSC32      cpu(clk);

    integer cycle = 0;
    integer i;

    initial begin
        $readmemh("init.dat", cpu.m);

        cpu.rf[0] = 32'd0;
        cpu.rf[1] = 32'd0;
        cpu.rf[2] = 32'd0;
        [...]
        cpu.rf[15] = 32'd0;

        cpu.vrf[0] = 128'd0;
        cpu.vrf[1] = 128'd0;
        cpu.vrf[2] = 128'd0;
        [...]
        cpu.vrf[15] = 128'd0;

        cpu.pc = 0;
    end
endmodule

```

```

        #10000 $stop;
    end

    always begin
        #1 clk = 0;
        #1 clk = 1;

        cycle <= cycle + 1;

        $display("----- %d", cycle);
        $display("PC:    %h", cpu.pc);

        $display("Instructions:");
        $display(" p0 - %h - %h %h %h %h", cpu.instr0, cpu.op_0, cpu.rA_0, cpu.rB_0, cpu.rC_0);
        $display(" p1 - %h - %h %h %h %h", cpu.instr1, cpu.op_1, cpu.rA_1, cpu.rB_1, cpu.rC_1);

        $display("Scalar Register Contents:");
        $display(" r0 - %h", cpu.rf[0]);
        $display(" r1 - %h", cpu.rf[1]);
        $display(" r2 - %h", cpu.rf[2]);
        $display(" r3 - %h", cpu.rf[3]);
        [...]
        $display(" rF - %h", cpu.rf[15]);

        $display("Vector Register Contents:");
        $display(" v0 - %h", cpu.vrf[0]);
        $display(" v1 - %h", cpu.vrf[1]);
        $display(" v2 - %h", cpu.vrf[2]);
        $display(" v3 - %h", cpu.vrf[3]);
        [...]
        $display(" vF - %h", cpu.vrf[15]);
    end

endmodule

```

The module instantiates a copy of the RiSC32 module and feeds it a clock signal. It also prints out some of the RiSC32’s internal state—note the naming convention used to get at the RiSC32’s internal variables. Note that the “initial” block executes before all else, where it initializes the RiSC32’s registers and memory. The \$readmemh call tells the simulator to overwrite the memory system with the contents of the file “init.dat” ... and then the initial block tells itself to halt execution 1000 cycles into the future. This is to stop any runaway processes due to design bugs; you can set this to larger values if you like.

Running Your CPU

First, “tap cadenceIC6” to get access to the simulators. Then you can invoke your code this way:

```

verilog test.v RiSC.v

```

or

```

ncverilog test.v RiSC.v

```

The *ncverilog* simulator has a longer start-up time but executes much faster once it gets going. This is useful for long-running benchmarks.

Use the *a32.pl* assembler to generate the *init.dat* file (“a32.pl prog.s > init.dat”).

Submitting Your Project

Submit your files through the on-line *submit* facility. When you submit your code to me, I only want your *RiSC32.v* file (or whatever name you have given it). I do not need anything else. For example, I do not need your *test.v* file (I will use my own). Do not change any of the variable names within the *RiSC32.v* file, for hopefully obvious reasons.

Do not worry about creating code that is synthesizable or efficient; future projects will focus on those aspects of design. For this project, just get it to work.