Project 2: Pipelining (15%)

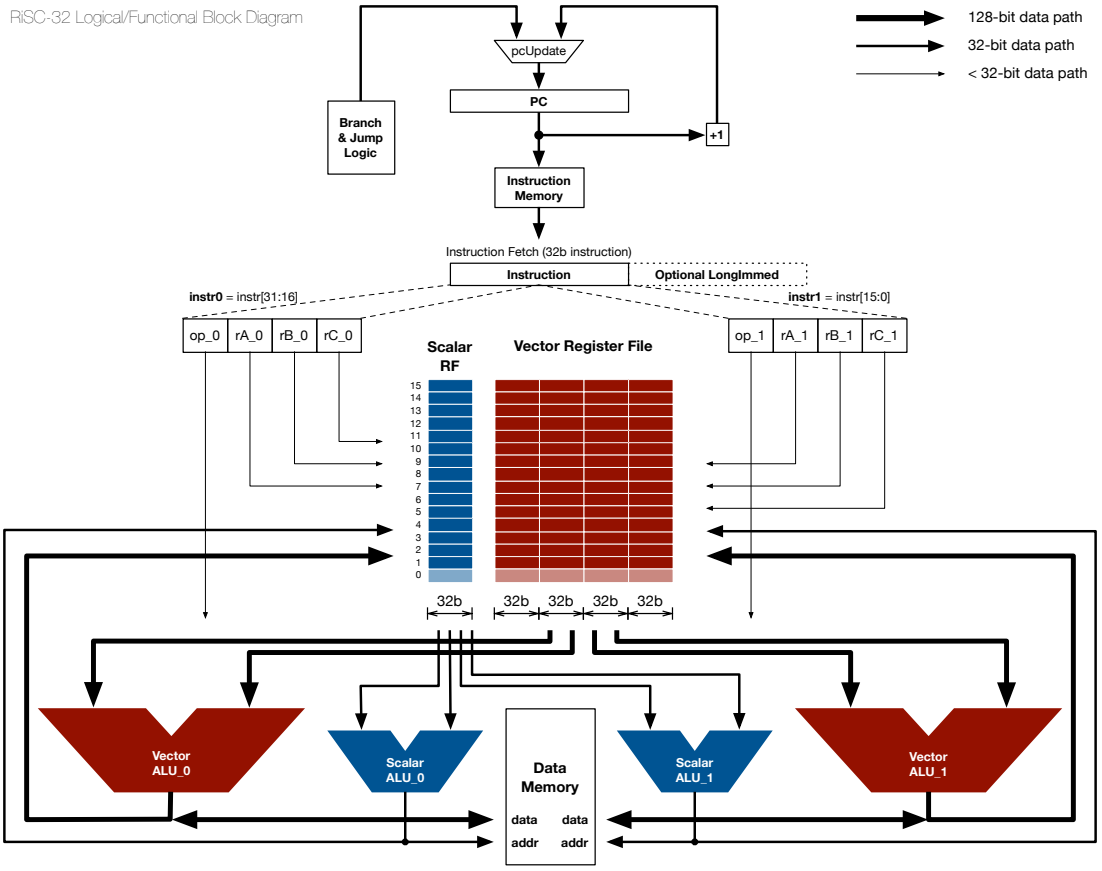
ENEE 446: Digital Computer Design, Spring 2018

Assigned: Wednesday, Feb 14; Due: Monday, Mar 12

Purpose

This project is intended to help you understand in detail how a pipelined microprocessor works. You have already looked at two different forms of *instruction-level parallelism* in the first project, in which you have a *VLIW (Very Large Instruction Word)* processor that can execute two instructions simultaneously, as well as a *vector* or *SIMD (Single-Instruction, Multiple-Data)* capability that can perform a single operation on four different data simultaneously. These are both examples of instruction-level parallelism, both fairly different in their approach. *VLIW* allows the compiler to express parallelism by grouping together independent atomic operations into a single monolithic instruction. *SIMD* allows the software to move and operate on data in large sequential chunks, which is why it is also called vector processing.

In this project, you will explore yet another form of instruction-level parallelism: *pipelining*, in which the microprocessor acts as an assembly line, operating on different phases of multiple different instructions, all simultaneously. You will build a pipelined RiSC-32, complete with simple *branch prediction*, realistic *data movement* and *forwarding*, and rudimentary *speculative execution*. In addition, we will be using a coding style in this project that is more *synthesizable* than the previous project, so you will also get a much better feeling for how data values and control signals are moved around the processor.



Pipelines

In the previous project, you built a sequential processor. In that example, the entire instruction is executed before the following clock edge, at which point the results of the instruction are latched into the register file and/or data memory, and the next instruction address is latched into the program counter. Not surprisingly, doing everything in a single clock results in a relatively long clock period, because a lot of things must happen all during the same clock cycle: the instruction is fetched, data values are read from the register file, ALU operations are performed on the data, perhaps memory is accessed, and any results generated are stored to the register file. That requires a long clock period, and long clock periods mean slow clock speeds.

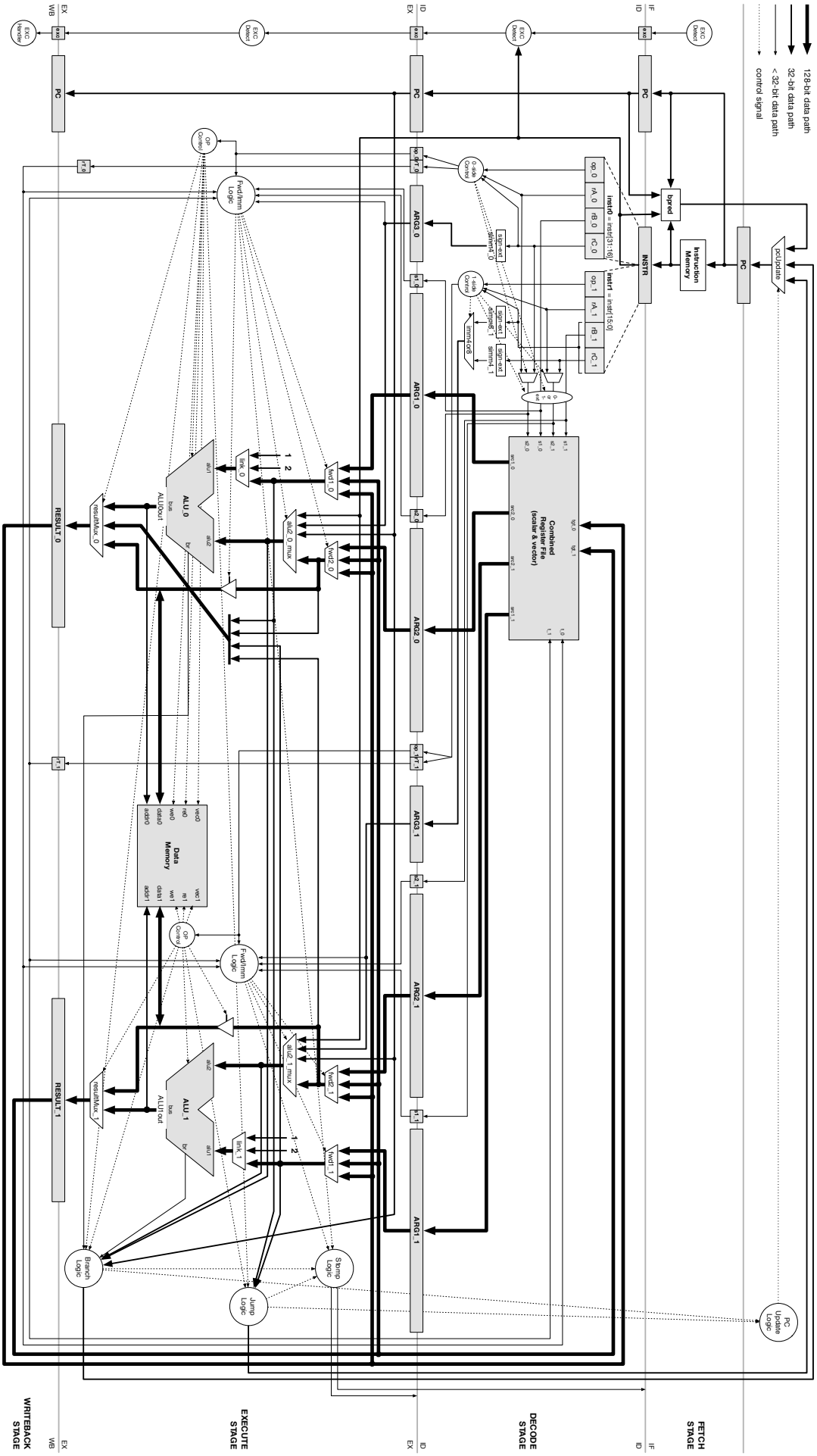
This is a problem, because the computer market is not fond of slow clock speeds: they imply low or inferior performance. One can improve performance by increasing clock speed, and one can increase clock speed by decreasing the amount of logic between successive latches. The easiest way to do this is to cut the instruction logic into ever-finer pieces or *stages*: if execution is sliced up into smaller stages, the clock speed is limited by the longest stage. Theoretically, a pipeline of N stages should run with a clock that is N times faster than a sequential implementation. For many reasons, this theoretical limit is never reached, due to latch overhead, sub-tasks of unequal length, etc. Nonetheless, extremely fast clock rates are possible. Slicing up the instruction execution this way is called *pipelining*, and it is exploited to great degree in nearly every aspect of modern computer design, from the processor core to the DRAM subsystem, to the overlapping of transactions on memory and I/O buses, etc.

The RiSC-32 pipeline is shown on the next page. In the figure, shaded boxes represent clocked registers; thick lines represent 128-bit buses; medium lines represent 32-bit buses; thin lines represent smaller data paths; and dotted lines represent control paths. The figure illustrates how pipelining is achieved: the sub-tasks into which instruction execution has been divided are instruction *fetch*, instruction *decode*, instruction *execute* (which includes both ALU operations and memory access), and register-file *writeback*. Each of these sub-tasks, which is executed by dedicated hardware called a *pipeline stage*, produces intermediate results that must be stored before an instruction may move on to the next stage. By breaking up execution into smaller sub-tasks, it is possible to overlap the different sub-tasks of several different instructions simultaneously. If the intermediate results of the various sub-tasks are not stored, they would be lost: during the next cycle another instruction would use the same hardware for its own task. For instance, after an instruction is fetched, it is necessary to store the fetched instruction somewhere, because the output of the instruction memory will be different on the following cycle—the fetch stage will be fetching a completely different instruction.

The storage locations for the intermediate results are called *pipeline registers*, and the figure illustrates their contents. It is common to label a pipeline register by the two stages that it divides. Thus, the pipeline register that divides the instruction fetch (IF) and instruction decode (ID) stages is called the *IF/ID register*; the pipeline register that divides the instruction decode (ID) and instruction execute (EX) stages is called the *ID/EX register*; and the register that divides the instruction execute (EX) and writeback (WB) stages is called the *EX/WB register*.

RiSC-32 Pipeline Registers

To simplify the design, rather than maintaining two separate register files and separating the wiring at the topmost level, the two register files are combined into a unified module. Thus, one of the first thing that happens in the decode stage is that register identifiers, which are 4 bits in length (each specifies one of 16 registers), are appended with a 1-bit demarcation indicating whether the reference is to the *scalar register file* (0) or the *vector register file* (1). This is a purely internal demarcation, which enables several things: first, it allows the various register files to be combined into a single monolithic structure, which simplifies wiring and circuit design; second, it makes data forwarding *much* more straightforward (imagine how to solve the forwarding problem otherwise). Thus, the following descriptions talk about


128-bit data path
32-bit data path
< 32-bit data path
control signal
FETCH STAGE
DECODE STAGE
EXECUTE STAGE
WRITEBACK STAGE
Instruction Memory
INSTR
PC
pcUpdate
Ipred
instr0 = instr[31:16]
instr1 = instr[15:0]
Combined Register File (scalar & vector)
ALU_0
ALU_1
Data Memory
RESULT_0
RESULT_1
ARG1_0
ARG2_0
ARG3_0
ARG1_1
ARG2_1
ARG3_1
Fwd/Imm Logic
Branch Logic
Jump Logic
Stmp Logic
PC Update Logic

5-bit register specifiers, even though the specifiers in the instruction word are each 4 bits long.

Similarly, the opcode represents not just the instruction's opcode, but it also represents whether the instruction is a VMOV type, which spans the entire instruction word. Thus, the following descriptions talk about 5-bit opcode specifiers, even though the opcodes in the instruction word are each 4 bits long.

Program Counter The address of the instruction currently being fetched.

IF/ID Register:

- **PC** Contains the address of the instruction whose state is represented in this pipeline register. This is used by BRANCH and JUMP instructions and in handling pipeline interrupts.
- **EXC** In a real implementation, this indicates whether the **fetch stage** caused an exception or not. Here we ignore it.
- **INSTR** The instruction to execute, with left/right-side atomic components: each with opcode, rA, rB, rC, and immediate fields.

ID/EX Register:

- **PC** Contains the address of the instruction whose state is represented in this pipeline register. This is used by BRANCH and JUMP operations and in handling pipeline interrupts.
- **EXC** In a real implementation, this indicates whether either the **fetch stage** or the **decode stage** caused an exception, for either atom. In general, if a stage's incoming EXC value is non-zero, it is passed down, regular operation in that stage is disabled, and the instruction is dynamically turned into two NOPs for following stages.
- **OP_0/1** Contains each atom's 5-bit opcode. In most cases, the opcodes are independent, and the **op_0/1** fields simply reflect each **op** field of the corresponding atom, with a '0' value appended. However, in the case of a VMOV instruction, the opcodes are not independent, and the opcode in **instr1** is the actual operation to be performed. Thus, the **op_0/1** fields are 5 bits wide, where the extra bit indicates whether the opcode in **instr0** is a VMOV or not. The topmost bit is '1' if **instr0.op** is VMOV, and it is '0' otherwise. The bottom four bits equal the VMOV sub-opcode (**instr1.op**) if **instr0.op** is VMOV, and they equal the corresponding **instr0/1.op** otherwise.
- **rT_0/1** Contains each atom's 5-bit target-register identifier, or the 5-bit binary value 00000 if the atom has no target (e.g., STORE and BRANCH operations). The top bit is '0' if the target is the scalar register file; the top bit is '1' if the target is the vector register file.
- **ARG3_0/1** Contains each atom's 32-bit immediate operand. If the atom uses a sign-extended immediate value (ADDI, LOAD, STORE, BRANCH), that value is available immediately and is stored here. For **inst0** (the left-side atom), the 4-bit value in rCi is always extended and placed here, whether the atom uses it or not. For **inst1** (the right-side atom), if the opcode is a BRANCH, the 8-bit value that spans rB and rCi is extended and placed here; otherwise, the 4-bit value in rCi is extended and placed here as described for inst0.
- **S1_0/1** Contains the 5-bit register specifier for each atom's **rB** argument; i.e., this identifies for each atom what register its first argument wants to read from and is used for data forwarding within the EXECUTE stage. The top bit is '0' if the specifier references the scalar register file; the top bit is '1' if the specifier references the vector register file.
- **ARG1_0/1** Contains the first register operand, called **rB** within the atom; this is the contents of the register *register-file[rB]*. This is either a 32-bit operand (if the corresponding s1 field contains the value 0xxxx) or a 128-bit operand (if the corresponding s1 field contains the value 1xxxx).
- **S2_0/1** Contains the 5-bit register specifier for each atom's **rC/rA** argument; i.e., this identifies for each atom what register its second argument wants to read from (whether rC or rA) and is used for data forwarding within the EXECUTE stage. The top bit is '0' if the specifier references the scalar register file; the top bit is '1' if the specifier references the vector register file.
- **ARG2_0/1** Contains the second register operand, called **rCi** within the atom. For most operations, it is the contents of *register-file[rC]*. For BRANCH, STORE, and JUMP operations, it is the contents of *register-file[rA]*. This is either a 32-bit operand (if the corresponding s2 field contains the value 0xxxx) or a 128-bit operand (if the corresponding s2 field contains the value 1xxxx).

EX/WB Register:

- **PC** Contains the address of the instruction whose state is represented in this pipeline register. This is used to handle pipeline interrupts.
- **EXC** In a real implementation, this indicates whether the **fetch stage** or the **decode stage** or the **execute stage** caused an exception, for either atom. In general, if a stage's incoming EXC value is non-zero, it is passed down, regular operation in that stage is disabled, and the instruction is dynamically turned into two NOPs for following stages.

  The **writeback stage** handles the exception if the value is non-zero. The only exception we will handle is HALT.
- **rT_0/1** Contains each atom's 5-bit target-register identifier, or the 5-bit binary value 00000 if the atom has no target (e.g., STORE and BRANCH operations).
- **RESULT_0/1** Contains the data that will be written to the register file on the following cycle (provided that the atom's corresponding **rT** field has a non-zero value).

Pipeline Behavior

The pipeline works on four separate instructions simultaneously and thus can have a clock that is roughly 4x faster than that of Project 1 (recall that in the Project 1 implementation, all aspects of instruction execution must be finished by the end of the cycle). The pipeline is divided into four stages, each of which performs a separate aspect of instruction execution at the same time as the others.

Fetch Stage & PC Update (including Branch Prediction)

The fetch stage uses the program counter (PC) to load a single 32-bit instruction from the memory system. Its primary input is the PC, and its primary output is the contents of the IF/ID register, described above. Note that the PC, along with any exceptional condition encountered, is passed down the pipeline along with the instruction being executed. This serves three distinct purposes:

1. The Program Counter allows branch mispredictions to be corrected later in the pipeline.
2. The Program Counter supports jump instructions that need to store the return target (PC+1) in the register file.
3. Together the Exception Code and the Program Counter enable support for precise interrupts, which allows the Operating System to be invoked if required (the Exception Code indicates what the OS should do), and because handling an interrupt involves jumping into the OS, this process requires that a return point is stored, which is what the Program Counter is used for.

Also included in this section is the update of the program counter, including the design of the branch predictor. PC update is the decision of which value to put into the program counter for the following cycle. There are several possibilities:

Branch Predictor Output	The **branch predictor** looks at the incoming instruction and scans it for BRANCH opcodes, which can be in either **instr0** or **instr1**. If either of the opcodes is a BRANCH, and its corresponding displacement is negative (i.e., a '1' bit is found in **instr0[3]**, or a '1' bit is found in **instr1[7]**), the sign-extended displacement is sent to the adder. Otherwise, the default PC update is used, which is a '1' value **even if** the instruction uses a **large immediate value**. The following explains. The **interesting part** comes when the **displacement field is all zeroes**, which indicates that the BRANCH uses a **large immediate value**, which is in the following word in memory. That means the immediate value ***has not been fetched yet***, and so the branch prediction must not happen until the following cycle. So the prediction has to be to continue fetching along this path (we always want to fetch the large immediate; it is never jumped over). **Thus, the operation of the branch predictor is as follows**. The logic preferentially looks at the instruction in the IF/ID register. If that instruction contains a BRANCH with a large immediate, the predictor makes its decision based on the currently-fetched 32-bit immediate value and, if the branch is predicted taken (has a negative offset), adds that to the PC in the IF/ID register. If the instruction in IF/ID has no branches, or branches that have small immediate values, then it is ignored (already predicted), and the currently-fetched instruction is examined. If that contains a predicted-taken branch, the current PC and currently-fetched offset is used.
Branch Logic Output	If a branch is **mispredicted** in the FETCH stage, then this fact is determined in the EXECUTE stage, at which point, the hardware must patch things up. The correct PC is computed, using the PC that has been passed down through the pipeline. If the branch is a **forward branch** (positive displacement), then it was predicted non-taken, and therefore the corrected PC would be found by adding the instruction's displacement, found in the **ARG3** field, to the PC stored in ID/EX. If the branch is a **backward branch** (negative displacement), then it was predicted taken, and therefore the corrected PC would be found by adding '1' to the PC stored in ID/EX if the branch uses a short displacement or '2' to the PC if the branch uses a long-form displacement.
Jump Logic Output	If the instruction in the ID/EX register is a JUMP instruction, then it updates the PC from a register in the register file. It also stores PC+1 into the register file as well.

Precedence between these outputs is described in the section below on *PC Update Logic*.

Decode Stage

The decode stage basically sets up the execution stage: it determines what data the operation will require and puts it into the correct place so that the data can be used on the following cycle. Each instruction requires three pieces of data:

- one value representing the instruction's sign-extended immediate value
- two values read from the register file

The sign-extended immediate value comes from either the least significant 8 bits (in the case of a BRANCH in **instr1**) or the least significant 4 bits (in all other cases). This value is stored as ARG3 in the ID/EX register. Note that it is safe to produce this even if the instruction does not use an immediate value (e.g., an ADD instruction). On the following cycle, the ARG3 value would simply go unused if it were unneeded.

The register-file values are specified by the **rB** value in the instruction and, depending on the opcode, either the **rA** value (in the case of BRANCHES or STORES) or the **rC** value (all other cases). The **rB** value corresponds to the register file's **src1** output; the **rA/rC** value corresponds to the register file's **src2** output. These are stored as ARG1 and ARG2 in the ID/EX register. And, as with the ARG3 value, if the instruction does not use both values produced by the register file, it is still safe to produce and store them: any unneeded value would simply go unused.

Note that the register file output can be either 32 bits or 128 bits (ARG1 and ARG2 are each 128 bits wide), depending on whether the instruction reads the scalar register file or the vector register file. The choice of register file is determined by the opcode. The decode stage must read the instruction's two opcodes and extend all of the instruction's 4-bit register specifiers to indicate whether each corresponds to a scalar quantity or a vector quantity. This indicates how many bits out of the ARG1 and ARG2 registers are valid. If the topmost bit of the extended register specifier is a '0' then the value is a 32-bit quantity; if the topmost bit of the extended register specifier is a '1' then the value is a 128-bit quantity.

The decode stage also determines the register targets for the instructions. The values for **rT** (one each for **instr0** and **instr1**) are stored in the ID/EX register, and then they are passed to the writeback stage, where they determine where the instruction writes its results. Accordingly, if the instruction does not write the register file, then the value should be '0' (the 0th register is read-only). Thus, the **rT** values are 5 bits wide, not 4, and the top bit indicates whether the instruction writes the scalar register file or the vector register file.

The decode stage also produces the **op_0/1** values that are stored in the ID/EX register to be used during execute; these values are also 5 bits wide, not 4 bits: they have an additional bit indicating VMOV status. In in most cases the **op_0/1** fields in the ID/EX register will simply be the corresponding atom's opcode, and the extra top bit is a '0' value. In the case that **instr0.op** is VMOV, then the top bit of each **op_0/1** field is a '1' value, and the bottom 4 bits are the opcodes (and, for **instr1**, in this case, it contains the CTL sub-opcode: '0' for VEC, '1' for VLO, and '2' for VHI).

An **op** of '0' together with an **rT** of '0' indicates a **nop** instruction.

Lastly, the decode stage handles the one instance of exceptional conditions that we will implement in this project: the HALT instruction. It scans the incoming instruction and writes the **IDEX.exc** register. If the incoming instruction has a JALR instruction, the instruction's immediate bits (the four **rCi** bits) are put verbatim into the **IDEX.exc** register. If the instruction contains **two** JALR instructions, then a non-zero value wins over a zero value.

Execute Stage

The execute stage is the heart of the pipeline and does most of the heavy lifting. Its primary input is the ID/EX register, and its data output is the contents of the EX/WB register. It also has control output that determines the next state of the Program Counter, as well as the IF/ID and ID/EX registers.

Due to the fact that this is a VLIW pipeline (there are multiple parallel pipelines, supporting multiple instruction issue), the data forwarding within the pipeline is already more complex than that of a single-issue pipeline by a factor of n^2. Thus, two design decisions were made to offset this as much as possible. First, all aspects of execution occur in a single cycle: namely, load/store operations are not divided into separate *address-generation* and *memory-access* stages but are instead combined in the same stage. Thus, unlike the classic 5-stage MIPS pipeline, the RiSC-32 4-stage pipeline has no need to stall on load-use interlocks (indeed, it does not stall at all, except for cache misses), and its data-forwarding is reduced by one stage of comparisons. Second, branch resolution is done in the execute stage and not at the end of the decode stage. This also reduces data-forwarding by one stage of comparisons. The net result is that, as compared to the MIPS pipeline and RiSC-16 pipeline, which have **three** stages of comparison (to perform data forwarding, one must look at the **three** instructions ahead of the current instruction), the RiSC-32 pipeline only looks **one** stage ahead. The disadvantage of this is that the branch/jump penalty is increased to two clock cycles. The branch prediction described above addresses the branch-penalty issue. One could install a branch-target buffer to address the jump penalty as well.

The data operations are carried out by two SIMD ALUs and the data memory. The ALU inputs are two 128-bit data busses and the function to be performed on them. Note that vector operations will be full 128-bit operations, and scalar operations will only use the least significant 32 bits of each 128-bit bus. The ALU-function input determines how much data is processed. Similarly, each ALU has a 128-bit data-output bus, and sometimes only 32 bits of that bus will be valid.

The data memory has two independent ports; each port takes as input a 16-bit *data address* (the memory is 256KB total), a *read-enable* signal (indicates whether or not IDEX contains a *load* instruction), a *write-enable* signal (indicates whether or not IDEX contains a *store* instruction), and a control signal indicating whether the I/O operation is a *vector* operation (128-bit) or a *scalar* operation (32-bit). Each port has a 128-bit *data* input that is bi-directional (outputs data on load operations; reads data in on store operations).

The ultimate output of the **execute stage** is four-fold:

1. Result data produced by the ALUs and/or data memory is stored in the 128-bit *result* fields of the EX/WB pipeline register, along with the 5-bit target-register specifiers that determine where, if anywhere, that data is to be stored on the following cycle.
2. Branch mispredictions and jump instructions cause the execute stage to override the PC Update process.
3. Branches and jumps also cause the IF/ID and ID/EX registers to be zeroed out as **nops**.
4. Similarly, instructions that may have large immediate values stored in the following memory address (ADDI and BRANCH instructions are allowed to do so) will cause the ID/EX register to become zeroed out as two **nops** whenever a large-immediate value is in fact used. When the instruction is in the execute stage, its large immediate value is found in the 32-bit **instr** field of the IF/ID pipeline register. It is important that this 32-bit value not be interpreted as an instruction, and so in these instances, the output of the decode stage is nullified, and zeroes are stored in the ID/EX register, signifying **nop** instructions.

The multiplexers of the stage manage the flow of data in and between the ALUs and data memory, and their operation is described below for each instruction. The following describes these operations:

Scalar Instructions:

add 0

The **fwd** muxes select whichever register value is the most recent. If either of the **rT** fields in the EX/WB register matches either of the **s1** or **s2** fields of the ID/EX register, then take the corresponding data from the EX/WB register. Otherwise, select the data from the **arg1** or **arg2** field of ID/EX.

The **link** mux selects the output of the **fwd1** mux.

The **alu2** mux selects the output of the **fwd2** mux, because the instruction does not use an immediate value.
The **resultMux** mux chooses the output of the ALU to send to the EX/WB register.

All data busses should have the bottom 32 bits valid.

addi 1

The **fwd1** mux selects whichever register value is the most recent. If either of the **rT** fields in the EX/WB register matches the **s1** field of the ID/EX register, then take the corresponding data from the EX/WB register. Otherwise, select the data from the **arg1** field of ID/EX.

The **link** mux selects the output of the **fwd1** mux.

The **fwd2** mux can be ignored because the second input is an immediate; the **alu2** mux selects the appropriate immediate value (either the value stored in the **arg3** field of ID/EX, or the **instr** field of IF/ID if the bottom 4 bits of **arg3** are zero).

The **resultMux** mux chooses the output of the ALU to send to the EX/WB register.

All data busses should have the bottom 32 bits valid.

and 2

(same as **add**)

mul 3

(same as **add**)

sub 4

(same as **add**)

lw 5

The **fwd1** mux selects whichever register value is the most recent. If either of the **rT** fields in the EX/WB register matches the **s1** field of the ID/EX register, then take the corresponding data from the EX/WB register. Otherwise, select the data from the **arg1** field of ID/EX.

The **link** mux selects the output of the **fwd1** mux.

The **fwd2** mux can be ignored because the second input is an immediate; the **alu2** mux selects the output of the **arg3** field of ID/EX.

The **we** input of Data Memory should be '0' and the **vec** input of Data Memory should be '0'.

The bottom 32 bits of the ALU output are tied to the **address** input of the Data Memory.

The **resultMux** mux chooses the output of the Data Memory to send to the EX/WB register. The bottom 32 bits of the bus should be valid.

sw 6

The **fwd** muxes select whichever register value is the most recent. If either of the **rT** fields in the EX/WB register matches either of the **s1** or **s2** fields of the ID/EX register, then take the corresponding data from the EX/WB register. Otherwise, select the data from the **arg1** or **arg2** field of ID/EX.

The **fwd2** mux cannot be ignored even though the second input is an immediate, because STORE operations have three arguments. The output of the **fwd2** mux will be the data input to the Data Memory. The bottom 32 bits of the bus should be valid.

The **link** mux selects the output of the **fwd1** mux.

The **alu2** mux selects the output of the **arg3** field of ID/EX.

The **we** input of Data Memory should be '1' and the **vec** input of Data Memory should be '0'.

The bottom 32 bits of the ALU output are tied to the **address** input of the Data Memory.

The **resultMux** mux can be ignored because the **rT** field should be '0,' indicating no data is to be written to the register file.

bne 7

Left-side operation when the opcode is BRANCH ('7'):

The **fwd** muxes select whichever register value is the most recent. If either of the **rT** fields in the EX/WB register matches either of the **s1** or **s2** fields of the ID/EX register, then take the corresponding data from the EX/WB register. Otherwise, select the data from the **arg1** or **arg2** field of ID/EX.

The **link** mux selects the output of the **fwd1** mux.

The **alu2** mux selects the output of the **fwd2** mux, because the instruction's immediate value is used as an offset to the Program Counter and is not used for data comparison in the ALU.

The **resultMux** mux can be ignored because the **rT** field should be '0,' indicating no data is to be written to the register file.

blz 7

Right-side operation when the opcode is BRANCH ('7'):

The **fwd2** mux selects whichever register value is the most recent. If either of the **rT** fields in the EX/WB register matches the **s2** field of the ID/EX register, then take the corresponding data from the EX/WB register. Otherwise, select the data from the **arg2** field of ID/EX. The **alu2** mux selects the output of the **fwd2** mux.

The **fwd1** mux can be ignored because the ALU internally will compare the **alu2** input to zero.

The **link** mux selects the output of the **fwd1** mux.

The **resultMux** mux can be ignored because the **rT** field should be '0,' indicating no data is to be written to the register file.

jalr 15/0xF

In the DECODE stage the **rCi** field of the atom should get inserted into the **EXC** field of the EX/WB register.

The **fwd1** mux selects whichever register value is the most recent. If either of the **rT** fields in the EX/WB register matches the **s1** field of the ID/EX register, then take the corresponding data from the EX/WB register. Otherwise, select the data from the **arg1** field of ID/EX. The bottom 32 bits of the output of the **fwd1** mux are used by the Jump Logic to update the Program Counter.

The **fwd2** mux can be ignored because JALR does not have a second register argument. The **alu2** mux chooses the PC field of the ID/EX register, and in the ALU, this is incremented by 1 and stored to the scalar register file during writeback.

The **link** mux chooses either 1 or 2 based on whether the instruction (the atom next to the JALR) uses a large immediate value or not. ***For all other instructions, the* link *mux chooses the output of the* fwd1 *mux.***

The **resultMux** mux chooses the output of the **ALU** to send to the EX/WB register. The bottom 32 bits of the bus should be valid.

Vector Instructions:

vadd 8

The **fwd** muxes select whichever register value is the most recent. If either of the **rT** fields in the EX/WB register matches either of the **s1** or **s2** fields of the ID/EX register, then take the corresponding data from the EX/WB register. Otherwise, select the data from the **arg1** or **arg2** field of ID/EX.

The **link** mux selects the output of the **fwd1** mux.

The **alu2** mux selects the output of the **fwd2** mux, because the instruction does not use an immediate value.

The **resultMux** mux chooses the output of the ALU to send to the EX/WB register.

All data busses should have all 128 bits valid.

vsum 9

The **fwd1** mux selects whichever register value is the most recent. If either of the **rT** fields in the EX/WB register matches the **s1** field of the ID/EX register, then take the corresponding data from the EX/WB register. Otherwise, select the data from the **arg1** field of ID/EX. All 128 bits should be valid.

The **link** mux selects the output of the **fwd1** mux.

The **fwd2** and **alu2** muxes can be ignored because the instruction has no second argument.

The **resultMux** mux chooses the output of the ALU to send to the EX/WB register. Only 32 bits should be valid.

vand 10/0xA

(same as **vadd**)

vmul 11/0xB

(same as **vadd**)

vxor 13/0xC

(same as **vadd**)

vlw 12/0xD

The **fwd1** mux selects whichever register value is the most recent. If either of the **rT** fields in the EX/WB register matches the **s1** field of the ID/EX register, then take the corresponding data from the EX/WB register. Otherwise, select the data from the **arg1** field of ID/EX.

The **link** mux selects the output of the **fwd1** mux.

The **fwd2** mux can be ignored because the second input is an immediate; the **alu2** mux selects the output of the **arg3** field of ID/EX.

The **we** input of Data Memory should be '0' and the **vec** input of Data Memory should be '0'.

The bottom 32 bits of the ALU output are tied to the **address** input of the Data Memory.

The **resultMux** mux chooses the output of the Data Memory to send to the EX/WB register. All 128 bits of the bus should be valid.

vsw 12/0xE

The **fwd** muxes select whichever register value is the most recent. If either of the **rT** fields in the EX/WB register matches either of the **s1** or **s2** fields of the ID/EX register, then take the corresponding data from the EX/WB register. Otherwise, select the data from the **arg1** or **arg2** field of ID/EX.

The **fwd2** mux cannot be ignored even though the second input is an immediate, because STORE operations have three arguments. The output of the **fwd2** mux will be the data input to the Data Memory. All 128 bits of the bus should be valid. The **link** mux selects the output of the **fwd1** mux.

The **alu2** mux selects the output of the **arg3** field of ID/EX.

The **we** input of Data Memory should be '1' and the **vec** input of Data Memory should be '0'.

The bottom 32 bits of the ALU output are tied to the **address** input of the Data Memory.

The **resultMux** mux can be ignored because the **rT** field should be '0,' indicating no data is to be written to the register file.

VMOV Instructions:

vec 14/0xE +0

The **resultMux_0** mux (which in the Verilog is just the definition for **EXWB_rfdata_0__in**, the input to the **rfdata** field of the EX/WB register) selects an aggregation of values: the four outputs of the four **fwd** muxes are ganged together into a single 128-bit value and stored into the **rfdata** field of the EX/WB register. Note that the least-significant 32 bits of each MUX output are used.

vlo 14/0xE +1

The **resultMux_0/1** muxes (which in the Verilog are just the definitions for **EXWB_rfdata_0/1__in**, the inputs to the **rfdata_0/1** fields of the EX/WB register) select the **bottom half** (least significant 64 bits) of the 128-bit vector output of the **fwd1** muxes (which both read the same register). The **resultMux_0** mux selects the least significant word of the vector, and the **resultMux_1** mux selects the next-least significant word of the vector.

vhi 14/0xE +2	The **resultMux_0/1** muxes (which in the Verilog are just the definitions for **EXWB_rfdata_0/1__in**, the inputs to the **rfdata_0/1** fields of the EX/WB register) select the **top half** (most significant 64 bits) of the 128-bit vector output of the **fwd1** muxes (which both read the same register). The **resultMux_1** mux selects the most significant word of the vector, and the **resultMux_0** mux selects the next-most significant word of the vector.

Writeback Stage

The writeback stage has two responsibilities: first, it updates the register file; second, it handles exceptions. The register-file update is relatively simple: if an instruction uses a non-zero **rT** target specifier, then it writes the register file (e.g., there is no need for a separate *write-enable* signal). As mentioned earlier, the 4-bit register specifier is extended in the decode stage to a 5-bit specifier, wherein the top bit identifies whether the instruction writes a 128-bit value to the vector register file (the top bit of the **rT** target specifier is '1') or writes a 32-bit value to the scalar register file (the top bit of the **rT** target specifier is '0').

The exception handling is also simple: if the **EXWB.exc** register is non-zero, halt the processor.

Control Modules

These are the descriptions of the various CONTROL modules, which are represented in the diagram as circles. The list starts in the upper left-hand corner and moves counter-clockwise around the diagram.

EXC Control and EXC Handler	These modules are responsible for detecting error situations and reporting them to the operating system. Each module monitors the behavior of its stage and also considers the incoming **EXC** value. In general, if a stage's incoming **EXC** value is non-zero, it is passed down, regular operation in that stage is disabled, and the instruction operation is dynamically turned into two NOPs for following stages. In real processors, this facility is the heart of the machine's ability to run operating systems. In this project, we will only use it to handle HALT instructions. In the **decode stage**, the module scans the incoming instruction and writes the **IDEX.exc** register. If the incoming instruction has one or more JALR instructions, the instruction's immediate bits (the four **rCi** bits) are put verbatim into the corresponding **IDEX.exc** register. In the **writeback stage**, a non-zero **EXC** value indicates that the processor should shut down.
0/1-side Control	These modules control the RF muxes that decide which of rC or rA to use as a register specifier (for BRANCH, STORE, and JUMP instructions, rA is used; for all others, rC is used), and they also extend the read- and write-register specifiers with 0 or 1 based upon whether the instruction reads/writes the scalar register file (extend with a '0') or reads/writes the vector register file (extend with a '1'). Note that a very few operations interact with *both* register files, such as the VSUM operation, which reads a single vector from the vector register file and writes a scalar result to the scalar register file, and the VEC operation, which reads four scalar values from the scalar register file and writes them as a single vector to the vector register file.
Op Control	These modules perform the tasks that are opcode-specific and are independent of the instruction's other atom. The module determines the FUNCalu signal that tells the ALU what operation to perform. We have simplified the ALU input so that it takes simply the opcode, and it determines internally what operation to perform. The module also drives the control signals of the Data Memory port, including the Write-Enable signal (if the opcode is LOAD), and the Vector signal, which indicates whether the LOAD or STORE value is a 32-bit scalar (VEC=0) or a 128-bit vector (VEC=1).
Fwd/Imm Logic	The module handles data forwarding by driving the two MUXes that determine the ALU's input. For the **ALU1** input, the choices are the Program Counter (for JUMP instructions), the ARG1 output from the previous stage (value read from the register file), or one of the two RESULT outputs of the previously executed atoms, which are currently stored in the EX/WB register. For the **ALU2** input, the choices are the 32-bit sign-extended immediate value stored in ARG3, the 32-bit large-immediate value found in the instruction word immediately following the present one (which, at this point, is the 32-bit value held in the INSTR field of the IF/ID pipeline register), the ARG2 output from the previous stage (value read from the register file), or one of the two RESULT outputs of the previously executed atoms, which are currently stored in the EX/WB register.

Branch Logic The branch-prediction logic in the FETCH stage scans the incoming instruction, and if it finds a backwards branch (opcode is BRANCH; top bit of displacement is '1'), then it predicts that branch to be taken. Any other branch is predicted to be non-taken. This module in the execute stage verifies whether or not that prediction was correct. Thus, if it is seen that a backwards branch was in either atom, and it was determined to be NOT TAKEN, or if a forward branch was in either atom, and it was determined to be TAKEN, then this module indicates to the STOMP and PC UPDATE logic blocks that this is the case. It uses the Program Counter and either the value '1' or the displacement held in the ARG3 field of ID/EX to generate the correct program counter value.

If there are two branches in the instruction, then this logic prioritizes only if both are determined to be taken. If both are non-taken, then it is as if the instruction has but one branch. If one branch indicates TAKEN and the other indicates NOT-TAKEN, then the TAKEN path is chosen, regardless of which atom produces the result. If both atoms have TAKEN branches, then the left-side atom (instr0) takes precedence.

Jump Logic This module determines if a JUMP operation is in the instruction, or if the operation contains an EXTENDED operation such as HALT (a JUMP instruction with a non-zero rCi value). If the instruction contains two JUMP operations, and both try to update the PC, then the left-most (instr0) takes precedence. Otherwise, the one updating the program counter is active.

If the instruction contains a HALT operation, the logic sets the X field of the EX/WB pipeline register, so that in the Writeback stage the HALT can take affect.

Stomp Logic This module controls the STOMP logic: it handles branch mispredictions and JUMP operations. The module's inputs are the outputs of the BRANCH and JUMP modules above. If either a JUMP or a branch misprediction happens, the two instructions following the JUMP/BRANCH are canceled (turned into NOP instructions: 32-bit '0' values).

PC Update Logic This module controls the operation of the **pcUpdate** mux. It handles branch mispredictions and JALR instruction execution. The module's inputs are the Branch and Jump modules. If both are active, i.e. both contain an atom that is trying to redirect the PC, then the prioritization is as follows: BRANCH TAKEN takes precedence; JUMP is next; BRANCH NOT TAKEN is last. So if one instruction is a mispredicted TAKEN branch, and the other is a JUMP, the BRANCH takes precedence. If one instruction is a mispredicted NON-TAKEN branch, and the other is a JUMP, the JUMP takes precedence.

RiSC-32 Instruction Set

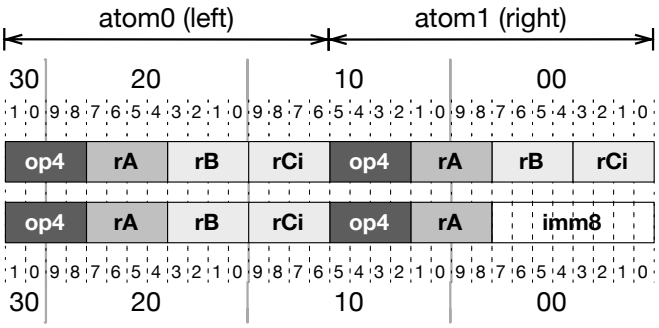
Here we reproduce the Instruction-Set information from Project 1. Note that **NAND/VNAND** instructions are now **AND/VAND**. The various operations are explained in the tables below.

Scalar Operations

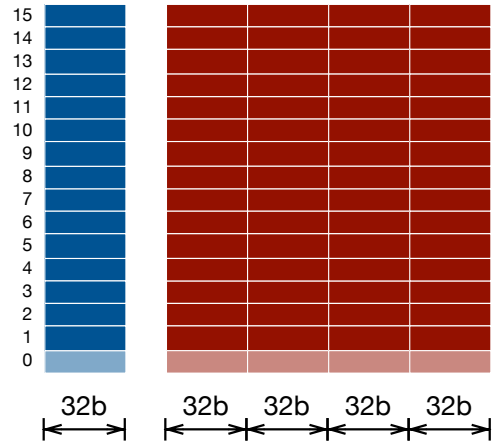
Inst Opcode	Assembly Format	Action	Verilog Pseudocode
add 0	add rA, rB, rC	Add contents of regB with regC, store result in regA.	`R[rA] <= R[rB] + R[rC]`
addi 1	addi rA, rB, imm	Add contents of regB with imm, store result in regA.	`R[rA] <= R[rB] + sign-extend imm4`
and 2	and rA, rB, rC	AND contents of regB with regC, store results in regA.	`R[rA] <= R[rB] & R[rC]`
mul 3	mul rA, rB, rC	Multiply contents of regB with regC, store result in regA.	`R[rA] <= R[rB] * R[rC]`
sub 4	sub rA, rB, rC	Subtract contents of regB from regC, store result in regA.	`R[rA] <= R[rB] – R[rC]`
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] <= m[ R[rB] + 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] => m[ R[rB] + 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.	`if ( R[rA] != R[rB] ) {`
			`  PC <= PC + sign-extend imm4`
			`} else {`
			`  PC <= PC + 1 (or 2 if imm4==0)`
			`}`

VLIW Instruction Format



Scalar RF / Vector Register File



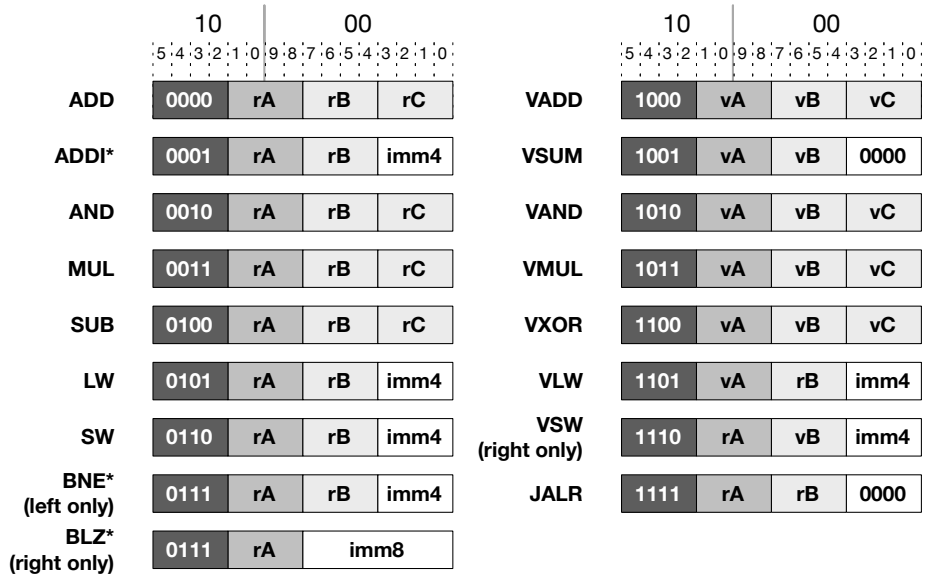
Big Endian: 0 1 2 3
Little Endian: 3 2 1 0

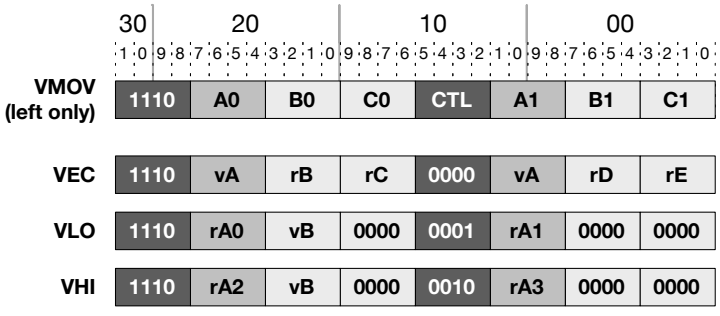
Opcodes:

0000 ADD	1000 VADD
0001 ADDI*	1001 VSUM
0010 AND	1010 VAND
0011 MUL	1011 VMUL
0100 SUB	1100 VXOR
0101 LW	1101 VLW
0110 SW	1110 VMOV/VSW
0111 BNE*/BLZ*	1111 JALR

*** imm=0 in ADDI or BRANCH**
=> next word is 32-bit immed

Atom Formats





CTL Values:
00 VEC
01 VLO (bottom 2)
10 VHI (top 2)

Inst Opcode	Assembly Format	Action	Verilog Pseudocode

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.	`if ( R[rA] < 0 ) {`
			`  PC <= PC + sign-extend imm8`
			`} else {`
			`  PC <= PC + 1 (or 2 if imm8==0)`
			`}`

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 <= R[rB]` `R[rA] <= 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] <= V[vB.0] + V[vC.0]`
			`V[vA.1] <= V[vB.1] + V[vC.1]`
			`V[vA.2] <= V[vB.2] + V[vC.2]`
			`V[vA.3] <= V[vB.3] + V[vC.3]`

vsum 9	vsum rA, vB	Sum the four 32-bit values in vecB, store results in scalar regA1.	`R[rA] <= V[vB.0] + V[vB.1] + V[vB.2] + V[vB.3]`

vand 10/0xA	vand vA, vB, vC	AND contents of vecB with vecC, store results in vecA.	`V[vA.0] <= V[vB.0] & V[vC.0]`
			`V[vA.1] <= V[vB.1] & V[vC.1]`
			`V[vA.2] <= V[vB.2] & V[vC.2]`
			`V[vA.3] <= 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] <= V[vB.0] * V[vC.0]`
			`V[vA.1] <= V[vB.1] * V[vC.1]`
			`V[vA.2] <= V[vB.2] * V[vC.2]`
			`V[vA.3] <= 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] <= V[vB.0] ^ V[vC.0]`
			`V[vA.1] <= V[vB.1] ^ V[vC.1]`
			`V[vA.2] <= V[vB.2] ^ V[vC.2]`
			`V[vA.3] <= V[vB.3] ^ V[vC.3]`

vlw 12/0xD	vlw vA, rB, imm	Load 128-bit value vecA from memory. Memory address is formed by adding imm with regB.	`V[vA] <= m[ R[rB] + sign-extend imm4 ]`
vsw 14/0xE	vsw vA, rB, imm *(right side only)*	Store 128-bit value vecA to memory. Memory address is formed by adding imm with regB.	`V[vA] => m[ R[rB] + sign-extend imm4 ]`

vec 14/0xE	vec vA, rB, rC, rD, rE *(full 32-bit word)*	Read four values from the scalar register file (rB, rC, rD, rE), write into the vector register file at register vecA	`V[vA.0] <= R[rB]`
			`V[vA.1] <= R[rC]`
			`V[vA.2] <= R[rD]`
			`V[vA.3] <= R[rE]`

vlo 14/0xE	vlo rA0, rA1, vB *(full 32-bit word)*	Read 0th and 1st scalars in vecB, store in scalar regA0 and regA1	`R[rA0] <= V[vB.0]` `R[rA1] <= V[vB.1]`
vhi 14/0xE	vhi rA2, rA3, vB *(full 32-bit word)*	Read 2nd and 3rd scalars in vecB, store in scalar regA0 and regA1	`R[rA2] <= V[vB.2]` `R[rA3] <= V[vB.3]`

Endianness

The question of *endianness* comes up when dealing with vectors. A vector that has sequential 32-bit values of 1, 2, 3, 4 … will have the following layout in memory:

```
000: 00000001 00000002 00000003 00000004
004: 00000005 00000006 00000007 00000008
008: 00000009 0000000a 0000000b 0000000c
012: 0000000d 0000000e 0000000f 00000010
016: 00000011 00000012 00000013 00000014
020: 00000015 00000016 00000017 00000018
```

What will be the result of the following instruction?

```
vlw v1, r0
```

This instruction treats the first four words of memory as a short 4-word array (four words, from the 0th word in memory to the 3rd word in memory) and loads them into vector register v1. How it does so can make a significant difference. There are two ordering options when these four words of memory are brought into register v1, corresponding to "big endian" and "little endian," which computer engineers have argued over for decades. These correspond to whether the 0th location is considered the high-order word of the vector or the low-order word. Is the vector in memory stored such that the most significant word comes first, or last?

Here are the two options, as they would be viewed in the 128-bit register file:

```
v1 - 00000001000000020000000300000004 [big endian]

v1 - 00000004000000030000000200000001 [little endian]
```

As can be seen, little endian makes more mathematical sense (the low-order words in the array correspond to the low-order words in the vector register), and big endian is more easily read (the layout in memory "looks like" the layout in the register file, because we tend to print things left to right).

Due to its overwhelming support during the class vote :), we will implement a **little endian** design.

Some related details regarding **vec**, **vlo**, and **vhi** instructions. Assume the following:

```
r1 = 0x11111111
r2 = 0x22222222
r3 = 0x33333333
r4 = 0x44444444
```

Then we have the following behaviors for **vec**, **vlo**, and **vhi** instructions:

```
vec v1, r1, r2, r3, r4
->      v1 = 11111111222222223333333344444444

vec v1, r4, r3, r2, r1
->      v1 = 44444444333333332222222211111111

vec v1, r4, r3, r2, r1
vlo r11, r12, v1
->      r11 = 22222222, r12 = 11111111

vec v1, r4, r3, r2, r1
vhi r11, r12, v1
->      r11 = 44444444, r12 = 33333333
```

In other words, the registers are read left-to-right as most significant to least significant quantities.

Verilog Implementation

On the course website is a skeleton Verilog file that includes definitions for all data structures that you will need, both registers and busses. You will not need to define any new registers, but feel free to define as many new wires as you see fit (e.g., to more finely break down logic blocks). The skeleton file contains definitions for all of the grey blocks shown in Figure 1 (pipeline registers, register file, memory, and ALU), and it contains instantiations of many of the control lines and multiplexer outputs—however, for most it does not give their combinational-logic definitions.

All of the registers are built as modules, each with a clock input, data input and output, reset signal, and a write-enable control signal. Your Verilog code will not need any register assignments; i.e., you need not put any code whatsoever into the "always @(posedge clk)" block at the end of the program, which is simply there to halt processing.