ESE 545

Model Cell SPU

Daniel Roach

Overview

This model of the Cell SPU simulates the execution of a subset of the Cell SPU instructions defined in the ISA. 106 instructions are implemented, as defined in the document "Instruction Subset.xlsx". This model uses features exclusive to VHDL-2008, so expect compilation errors if earlier VHDL versions are used.

This report is structured by sections of the Cell SPU. Each section will describe the design of the component as it was implemented in VHDL. Most of these components have either already been tested in the deliverable submitted on March 21st, 2022 or are effectively tested by the 4x4 matrix multiplication program, so in the interest of preventing this report from becoming any longer, I chose to avoid providing redundant tests that provide little to no insight into my model. I have included a copy of this deliverable.

An attempt was made to isolate each feature into its own VDHL entity. A few components exist to combine a bunch of related subcomponents to contribute to the effort of avoiding signal clutter.

Signal Names

Signals are generally named according to the following format: [stage]_[pipe/instruction]_signalname. If there are multiple signals that serve an identical purpose across multiple stages (writeback enable signals, data ready signals, etc), they are combined into an std_logic_vector with the index determining the stage they correspond to. Again, this was done to help reduce signal clutter.

In the Instruction Fetch and Instruction Decode stages, the two instructions have not been routed to the even and odd pipes, so the instructions are labelled based on the order they appeared in the instruction memory. Since two instructions are fetched each cycle, Instruction 0 (labelled in signals as i0) is the former instruction of the pair, and Instruction 1 (i1) is the latter. The terms Instruction 0 and Instruction 1 will be used extensively throughout this report to describe the locations of these instructions.

On the last page of this report, I've included a hand drawn diagram of my Instruction Fetch stage. It may be helpful to reference as I explain my design later in this report.

Execution Pipes

The Cell SPU has seven execution pipes. The multiply-accumulate pipe, simple fixed 1 pipe, simple fixed 2 pipe, and byte pipe are all considered "even" pipes. The permute pipe, load/store pipe, and branch pipe are all considered "odd" pipes. Logic within the instruction decode stage will ensure that at maximum one instruction is being executed by the set of even pipes and at maximum one instruction is being executed by the set of odd pipes.

Multiply Accumulate

The multiply accumulate pipe is a seven-stage execution pipe that is responsible for performing floating point addition and subtraction, integer and floating-point multiplication, and

integer and floating point multiply accumulation. All computations are performed on floating point values, so for integer operations, they get cast to floating point values, have the computation performed, and have their results cast back into integer values. The hardware that performs this cast is also used to implement the integer-floating point conversion instructions. The pipeline stages perform the operations as follows:

Stage 1: Operand Selection – Different instructions have different sources for their operands, so stage 1 consists of multiplexers to select the data from the correct source based on the control signals provided from the decode logic.

Stage 2: Integer to Floating Point Cast – Each operand selected in Stage 1 gets cast into a floating-point value if the cast integer to floating point control signal is set. An additional control signal enables scaling of these cast values, as defined by the integer-floating point conversion instructions.

Stage 3: Multiplication – If the multiply control signal is set, the multiplicand and multiplier will be multiplied into the product field. If the multiply control signal is not set, the product is assigned the value of the multiplicand. The multiplier is necessary for the multiply and multiply-accumulate instructions but must not be used for the floating-point addition instructions and the conversion instructions.

Stage 4: Addition – If the add control signal is set, the product and addend are added or subtracted based on the "add mode" control signal and placed into the result field. If the add control signal is not set, the result is assigned the value in the product field.

Stage 5: Negation – The instruction "Floating Negative Multiply and Subtract" is the negated result of the "Floating Multiply and Subtract" instruction, so Stage 5 implements that negation.

Stage 6: Floating Point to Integer Cast – At Stage 6, if the instruction was a floating-point instruction, the result is ready to be forwarded to new instructions and is output here. If the instruction was an integer instruction, the integer must still be cast back to an integer before the result is ready, so this conversion hardware exists in stage 6.

Stage 7: Output Integer Result

Simple Fixed 1

The Simple Fixed 1 pipe is a two-stage pipe that is responsible for integer addition and subtraction, logical instructions, load immediate, count leading zeros, generation of select masks, sign extensions, and comparisons.

Stage 1: Operand Selection – Stage 1 consists of multiplexers to route the incoming data to the operands for Stage 2 depending on the control signals issued to the pipe.

Stage 2: Execution – Stage 2 contains all the hardware to perform the computation of the instruction. This is split into 7 different datapaths: arithmetic, logical, comparisons, sign extensions, load immediate, select, and count leading zeros. All these datapaths compute their

respective result, and the 3-bit mini opcode generated by the instruction decoder is used as a select signal for an 8 to 1 multiplexer that selects the appropriate computation.

Simple Fixed 2

The Simple Fixed 2 pipe is a three-stage pipe that is responsible for word shift and rotate instructions. SF2 is implemented differently from the multiply-accumulate and SF1 pipes, in which the result is computed in the first stage of the pipeline, and the computed result is passed through the pipeline and marked as ready at the third stage. The shift and rotate instructions are both implemented as a rotation, but a control signal is used to mask out the bits that are rotated if the instruction is a shift.

Byte

The Byte pipe is a three-stage pipe that implements four byte-oriented instructions: population count, byte average, absolute difference of bytes, and sum bytes into halfwords. There is no need to have an operand selection stage, since all byte operations have register sources. Much like the second stage of the SF1 pipe, the first stage of the byte pipe performs the computations for the four byte operations and sends them forward to the second stage. The second stage uses a multiplexer to select the correct operation, and the third stage outputs the result.

Permute

The permute pipeline is a three-stage pipe that implements shuffles, bit gathers, and quadword shifts and rotates. Much like SF2, the permute pipe computes its result in the first stage and sends that result down through the pipe. Even though the permute pipe has three stages, the data is not ready to be forwarded until it reaches the fourth stage.

Load/Store

The load store pipeline provides access to the data in the load store unit. The LS pipe performs quadword stores using d-form and x-form addressing modes. For some reason, even though I did not select the a-form instructions in my "Instructions Subset.xlsx" file, they are supported by my assembler and Cell SPU model, and are used for convenience in the 4x4 Matrix Multiplication program.

Stage 1: Operand Selection – Selects the operands based on the addressing modes.

Stage 2: Address Computation – Computes the sum of the operands. This is the target address of the load or store operation.

Stage 3: Load/Store Access – Accesses the Load/Store entity and performs the desired load or store operation. In this model, this occurs within one clock cycle.

Stages 4 through 6: No Operation – Nop stages were implemented to propagate the result of a load operation after stage 3 and make the data available at stage 6 to model the behavior of the Load/Store in the Cell SPU.

Branch

The branch pipe is also an execution pipe; however, it does not store its results back to the register file. The defined behavior for this branch pipe will be described later in this report. The branch pipe takes in the program counter of the branch instruction, the immediate value, and a source register, and determines whether the branch is taken or not. It also takes the branch prediction as an input and uses it to determine if the branch was mispredicted.

Data Forwarding

The data forwarding system is comprised of three main types of components: forwarding data selects, forwarding registers, and a priority select in the register file stage. The data forwarding registers propagate the results from the execution pipes throughout the remaining stages of the even and odd pipes. The forwarding data selects will select data from the pipe that has its writeback enable signal set. These selects are one of few components not implemented as a VHDL entity, but just a process in the structural architecture that connects all the CPU components.

```
-- even stage 3 data selection
even_s3_fwd_select: process(all)
begin
    if s23_fwreg_data_ready_out = '1' then
        s3_even_data <= s23_fwreg_data_out;
        even_data_ready(3) <= s23_fwreg_data_ready_out;
    elsif sf2_wb_en(3) = '1' then
        s3_even_data <= sf2_result;
        even_data_ready(3) <= sf2_wb_en(3);
    elsif byte_wb_en(3) = '1' then
        s3_even_data <= byte_result;
        even_data_ready(3) <= byte_wb_en(3);
    else
        s3_even_data <= (others => '0');
        even_data_ready(3) <= '0';
    end if;
end process even_s3_fwd_select;
```

The priority select in the register file stage is only non-trivial component in the forwarding process. It uses a VHDL if-then block to check each stage of execution if there is data ready to be forwarded. The select searches through the stages starting from the second stage through to the writeback stage. As soon as it finds a target register that matches a source operand, the priority select forwards this value and stops searching for matches, because the most recent values for a target will be found in the earlier stages of the pipeline.

```
-- even ra data forwarding
fwd_sel_even_ra_proc: process(all)
begin
    -- STAGE 2
    if (even_ra_index = s2_even_index) and (even_data_ready(2) = '1') then      -- forward from stage 2 (even)
        even_ra <= s2_even_data;
    -- STAGE 3
    elsif (even_ra_index = s3_even_index) and (even_data_ready(3) = '1') then       -- forward from stage 2 (even)
        even_ra <= s3_even_data;
    -- STAGE 4
    elsif (even_ra_index = s4_even_index) and (even_data_ready(4) = '1') then       -- forward from stage 2 (even)
        even_ra <= s4_even_data;
    elsif (even_ra_index = s4_odd_index) and (odd_data_ready(4) = '1') then     -- forward from stage 2 (odd)
        even_ra <= s4_odd_data;
    -- STAGE 5
    elsif (even_ra_index = s5_even_index) and (even_data_ready(5) = '1') then       -- forward from stage 2 (even)
        even_ra <= s5_even_data;
    elsif (even_ra_index = s5_odd_index) and (odd_data_ready(5) = '1') then     -- forward from stage 2 (odd)
        even_ra <= s5_odd_data;
    -- STAGE 6
    elsif (even_ra_index = s6_even_index) and (even_data_ready(6) = '1') then       -- forward from stage 2 (even)
        even_ra <= s6_even_data;
    elsif (even_ra_index = s6_odd_index) and (odd_data_ready(6) = '1') then     -- forward from stage 2 (odd)
        even_ra <= s6_odd_data;
    -- STAGE 7
    elsif (even_ra_index = s7_even_index) and (even_data_ready(7) = '1') then       -- forward from stage 2 (even)
        even_ra <= s7_even_data;
    elsif (even_ra_index = s7_odd_index) and (odd_data_ready(7) = '1') then     -- forward from stage 2 (odd)
        even_ra <= s7_odd_data;
    elsif (even_ra_index = wb_even_index) and (even_wb_en = '1') then           -- forward from wb stage (even)
        even_ra <= wb_even_data;
    elsif (even_ra_index = wb_odd_index) and (odd_wb_en = '1') then             -- forward from wb stage (odd)
        even_ra <= wb_odd_data;
    else                                                                        -- no forwarding
        even_ra <= even_ra_in;
    end if;
end process fwd_sel_even_ra_proc;
```

Instruction Decoder

The instruction decoder takes in an encoded instruction and outputs the corresponding register source and target indices and the control signals for each pipe based on the instruction format. Control signals are implemented as a bus. Multiple execution pipes share identical control signals, so there is a shared_ctrl output that contains signals to indicate operand sources, 3-bit pipe specific opcodes, 2-bit operand size control signals, and a 2-bit control for memory access modes. Not every pipe uses all these signals, but these are the signals shared by one or more of the pipes. Making these signals shared across all execution pipes again contributes to the goal of minimizing signal clutter and redundancy. The multiply-accumulate and simple fixed 1 pipes also have pipe-specific control signals, which are also generated by the decode logic and output to a signal labelled [pipe name]_ctrl. The control signal at index 0 is the writeback enable signal for that specific pipe. All the other pipes also receive their own dedicated writeback enable input. The instruction decode also generates a "registers used" output, which is to indicate whether the decoded instruction uses each of the source registers. This set of signals is used by the stall generation logic to determine if a stall should be generated when a source index matches with a target index. The defined stall behavior will be elaborated upon later in this report, which will explain the necessity of these signals.

This is a component that is thoroughly tested by any test program, so if all other programs are functioning properly, the instruction decoder can be assumed to be functional. I made a preliminary, non-exhaustive testbench to verify the basic functionality of a few instructions.

Hazard Detection

There are three types of hazards in an in-order pipelined CPU: structural hazards, data hazards, and control hazards. Regardless of what hazard exists, it must not happen that Instruction 1 gets executed before Instruction 0, because the Cell SPU is an in-order processor, and it would be catastrophic if exceptions were generated out of program order.

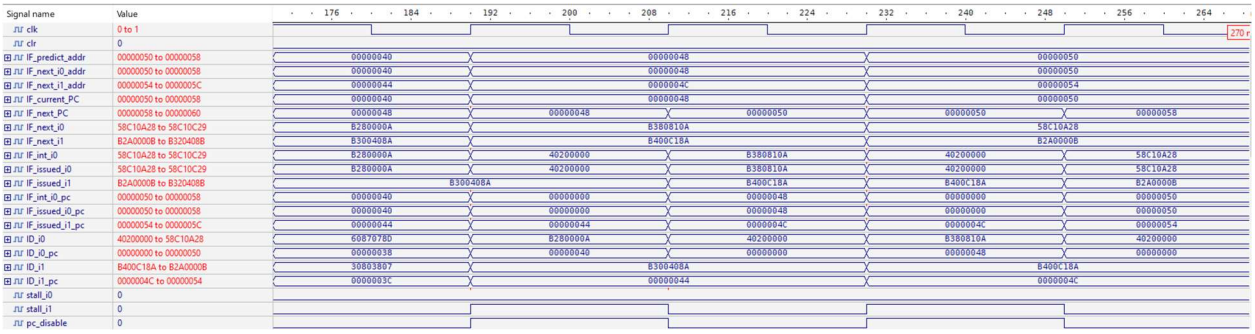
Structural Hazards

Structural hazards exist when two instructions both attempt to enter the same set of pipes (i.e. both enter the even pipe or both enter the odd pipe). The only way to resolve a structural hazard is to stall one of the two instructions attempting to enter the set of pipes. Since it was defined that Instruction 1 must not be executed before Instruction 0, Instruction 1 must get stalled.

Structural hazards are detected from the "structural hazard detector" component. To determine whether there is a structural hazard in the even pipe, the logical or of all the writeback enable signals for each instruction is computed, the logical and of those results is computed. The same is done for the odd pipe. If there is a structural hazard in either the even pipe or the odd pipe, there is a structural hazard, and the output of the structural hazard detector is set.

```
architecture dataflow of structural_hazard_detector is
signal even_hazard, odd_hazard  : std_logic;
begin
    even_hazard <= (i0_ma or i0_sf1 or i0_sf2 or i0_byte) and (i1_ma or i1_sf1 or i1_sf2 or i1_byte);
    odd_hazard <= (i0_perm or i0_ls or i0_br) and (i1_perm or i1_ls or i1_br);
    structural_hazard <= even_hazard or odd_hazard;
end dataflow;
```

To display the stall generator functioning, I've acquired a screenshot from the 4x4 SP FP matrix multiplication program over the interval 170 ns to 270 ns, in which four shuffle byte instructions appear consecutively in the program source code. It can be seen that, when two shuffle byte instructions are issued at the same time, Instruction 1 gets stalled when the stall_i1 signal is set, and the program counter stops incrementing until Instruction 1 gets issued. This happens twice because there are four shuffle byte instructions.



Data Hazards

A data hazard exists if the instruction in the hazard detection stage uses a source register that has a writeback pending in a stage of an execution pipe, and the result will not be ready for forwarding by the time this instruction reaches the forwarding logic. In this model, the hazard

detection occurs one cycle before data forwarding, so the data hazard detector must check all stages up to two cycles before the instruction will be ready from each pipe.

Not all instructions use at least one source register, but there is no way to undefine any of the source register indices, so the hazard detection needs to have a way to determine which source registers are actually going to be used by the instruction. I could either decode this information based off the control signals generated from each instruction, but I decided to just add a "register_used" signal for each of the registers. If the register_used signal is set, then the instruction is using that register, and stalls should be generated if a match is found. Without this logic, there is a chance that an instruction that uses no source registers, such as load immediate, would stall if another instruction in one of the execution pipes is going to write to the source register index that happens to be defined for the load immediate instruction. There is also the possibility that Instruction 1 is dependent on the result of Instruction 0, which also is a data hazard.

The outputs of the data hazard detector are two signals indicating whether a data hazard was found for Instruction 0 and Instruction 1. These will be used by a stall generator.

Below is the code to check if there is a data hazard for ra of Instruction 1. All of the other registers are check with an equivalent assignment.

```
i1_ra_hazard <=
-- check dependency on i0
((register_equal(i1_ra_index, i0_rt_index) and i0_wb_en) or
-- check RF stage
(register_equal(i1_ra_index, rf_even_index) and rf_even_wb_en) or (register_equal(i1_ra_index, rf_odd_index) and rf_odd_wb_en) or
-- check stage 1
(register_equal(i1_ra_index, s1_even_index) and (ma_int_wb_en(1) or ma_flt_wb_en(1) or sf2_wb_en or byte_wb_en)) or
(register_equal(i1_ra_index, s1_odd_index) and (perm_wb_en(1) or ls_wb_en(1))) or
-- check stage 2
(register_equal(i1_ra_index, s2_even_index) and (ma_int_wb_en(2) or ma_flt_wb_en(2))) or (register_equal(i1_ra_index, s2_odd_index) and (perm_wb_en(2) or ls_wb_en(2))) or
-- check stage 3
(register_equal(i1_ra_index, s3_even_index) and (ma_int_wb_en(3) or ma_flt_wb_en(3))) or (register_equal(i1_ra_index, s3_odd_index) and ls_wb_en(3)) or
-- check stage 4
(register_equal(i1_ra_index, s4_even_index) and (ma_int_wb_en(4) or ma_flt_wb_en(4))) or (register_equal(i1_ra_index, s4_odd_index) and ls_wb_en(4)) or
-- check stage 5
(register_equal(i1_ra_index, s5_even_index) and ma_int_wb_en(5))
) and i1_registers_used(1);
```

I have also acquired a screenshot from the 4x4 SP FP Matrix Multiplication program to illustrate a data hazard. Between 330ns and 430ns, the first set of FP multiplication operations are all issued over the course of four clock cycles, and the next set of FP multiply accumulate instructions can not be issued until the results from the previous batch are computed. In this case, both instructions are stalled because the multiply accumulate instruction is Instruction 0. Observe that stall_i0 is set for two clock cycles, and since stall_i0 is set, stall_i1 is also set. Since there is a stall, the program counters are disabled and do not continue until the stall is resolved.

Control Hazards

There is a control hazard every time a branch instruction is issued. The defined behavior will be explained in the "Branch" section.

Stall Generator

The stall generator is a simple piece of hardware that takes in the hazard signals as the inputs and outputs the appropriate stall signals. If there's a data hazard that requires Instruction 0 to stall, both Instruction 0 and Instruction 1 get stalled, since Instruction 1 can not be executed before Instruction 0. If there's a structural hazard or a data hazard for Instruction 1 and Instruction 0 can be issued, Instruction 0 is issued and Instruction 1 is stalled. If either instruction is stalled, the program counter is disabled, and the output to disable the program counter is set.

```
architecture dataflow of stall_generator is
begin
    stall_i0 <= i0_data_hazard;
    stall_i1 <= i0_data_hazard or i1_data_hazard or structural_hazard;
    pc_disable <= i0_data_hazard or i1_data_hazard or structural_hazard;
end dataflow;
```

Instruction Routing

The routing component is a simple device that takes in the control signals generated by the instruction decoders and routes the instructions to the even and odd pipes depending on the action specified by the stall generator. If both instructions are stalled, the router issues nop signals to the even and odd pipes. If only Instruction 1 is stalled, the router issues Instruction 0 to whichever pipe it needs to enter, and a nop to the other pipe. If both instructions can be issued, for each pipe the router checks if Instruction 0 is attempting to enter that pipe. If so, Instruction 0's control signals get issued to that pipe. Otherwise, it checks if Instruction 1 is attempting to enter that pipe. If so, Instruction 1's control signals are issued to that pipe.

Otherwise nop signals are issued to that pipe.

```
if (i0_ma_ctrl(0) = '1' or i0_sf1_ctrl(0) = '1' or i0_sf2_en = '1' or i0_byte_en = '1') then    -- i0 needs to be issued into the even pipe
    -- issue i0 even control signals to even instruction
    even_ra_index <= i0_ra_index;
    even_rb_index <= i0_rb_index;
    even_rc_index <= i0_rc_index;
    even_rt_index <= i0_rt_index;
    even_i16 <= i0_i16;
    even_shared_ctrl <= i0_shared_ctrl;
    even_ma_ctrl <= i0_ma_ctrl;
    even_sf1_ctrl <= i0_sf1_ctrl;
    even_sf2_en <= i0_sf2_en;
    even_byte_en <= i0_byte_en;
    even_pc <= i0_pc;
elsif (i1_ma_ctrl(0) = '1' or i1_sf1_ctrl(0) = '1' or i1_sf2_en = '1' or i1_byte_en = '1') then
    -- issue i1 even control signals to even instruction
    even_ra_index <= i1_ra_index;
    even_rb_index <= i1_rb_index;
    even_rc_index <= i1_rc_index;
    even_rt_index <= i1_rt_index;
    even_i16 <= i1_i16;
    even_shared_ctrl <= i1_shared_ctrl;
    even_ma_ctrl <= i1_ma_ctrl;
    even_sf1_ctrl <= i1_sf1_ctrl;
    even_sf2_en <= i1_sf2_en;
    even_byte_en <= i1_byte_en;
    even_pc <= i1_pc;
else    -- neither instruction was to use an even pipe, and there are no hazards, so issue an even nop
    even_ra_index <= (others => '0');
    even_rb_index <= (others => '0');
    even_rc_index <= (others => '0');
    even_rt_index <= (others => '0');
    even_i16 <= (others => '0');
    even_shared_ctrl <= (others => '0');
    even_ma_ctrl <= (others => '0');
    even_sf1_ctrl <= (others => '0');
    even_sf2_en <= '0';
    even_byte_en <= '0';
    even_pc <= (others => '0');
end if;
```

```
 if (i0_perm_en = '1' or i0_ls_en = '1' or i0_br_en = '1') then  -- i0 is to be issued to the odd pipe
     odd_ra_index <= i0_ra_index;
     odd_rb_index <= i0_rb_index;
     odd_rc_index <= i0_rc_index;
     odd_rt_index <= i0_rt_index;
     odd_i16 <= i0_i16;
     odd_shared_ctrl <= i0_shared_ctrl;
     odd_perm_en <= i0_perm_en;
     odd_ls_en <= i0_ls_en;
     odd_br_en <= i0_br_en;
     odd_br_pos <= '0';  -- if it's a branch instruction, it came from i0
     odd_pc <= i0_pc;
 elsif (i1_perm_en = '1' or i1_ls_en = '1' or i1_br_en = '1') then   -- i1 is to be issued to the odd pipe
     odd_ra_index <= i1_ra_index;
     odd_rb_index <= i1_rb_index;
     odd_rc_index <= i1_rc_index;
     odd_rt_index <= i1_rt_index;
     odd_i16 <= i1_i16;
     odd_shared_ctrl <= i1_shared_ctrl;
     odd_perm_en <= i1_perm_en;
     odd_ls_en <= i1_ls_en;
     odd_br_en <= i1_br_en;
     odd_br_pos <= '1';  -- if it's a branch instruction, it came from i1
     odd_pc <= i1_pc;
 else    -- neither instruction was to use the even pipe, and there are no hazards, so issue an odd nop
     odd_ra_index <= i0_ra_index;
     odd_rb_index <= i0_rb_index;
     odd_rc_index <= i0_rc_index;
     odd_rt_index <= i0_rt_index;
     odd_i16 <= (others => '0');
     odd_shared_ctrl <= i0_shared_ctrl;
     odd_perm_en <= i0_perm_en;
     odd_ls_en <= i0_ls_en;
     odd_br_en <= i0_br_en;
     odd_br_pos <= '0';
     odd_pc <= (others => '0');
 end if;
```

Branching, Branch Prediction, and Control Hazard Resolution

The Cell SPU ISA defines a branch by assigning the target address to the program counter if the branch is taken and incrementing the program counter by 4 if the branch is not taken. Since the Cell SPU fetches two instructions every clock cycle, each instruction has a different program counter value, and the value in the program counter register must be incremented by 8 every clock cycle. Furthermore, the branching instruction can be either Instruction 0 or Instruction 1, so if the branch is not taken and the branching instruction was Instruction 0, the value loaded into the program counter register must be the address of the branch instruction + 8, and if the branching instruction is Instruction 1, the value in the program counter register must be the address of the branch instruction + 4. Initially, I naively believed that this was a non-issue, as if the branch was not taken, the program counter would just continue incrementing without modification. However, it's possible that the branch prediction algorithm predicts a branch as taken, modifying the program counter register and fetching instructions from the predicted target address. If this branch is not taken, the branch pipe now must compute the next program counter value as PC+4 or PC+8.

Clearly, the necessary results for the next program counter computation are dependent on which instruction the branch came from, so I added a control signal called "branch position" (labelled in VHDL models as br_pos) to ensure the correct value is assigned to the program counter if a branch is not taken. Unfortunately, this is not the only behavior of the branching mechanism that is dependent on the position of the branch instruction, as will be seen when control hazard resolution is discussed.

In my design, the branch predictor is located after the instruction routing logic. This gives me the advantage of ensuring mutually exclusive occupation of the odd pipe, thus only requiring one branch predictor. The branch prediction algorithm implemented in my Cell SPU model is to predict all branches as taken. As such, the prediction algorithm simply computes the target address based on the addressing mode of the branch, and the predict_taken output is set if the instruction is a branch instruction. For a predicted branch, if the branch is predicted as taken, this new address is selected in the Instruction Fetch stage by the prediction multiplexer. If the instruction is not a branch or the prediction algorithm predicts the branch as not taken, the current program counter is selected. The greatest design flaw here is that this implementation imposes that the predict_taken output can only be set if the instruction is a branch instruction. As such, the branch predictor must have an input to indicate whether the odd instruction is a branch instruction and must clear the predict_taken output if the instruction is not a branch. In my opinion, this does not go well with my goal of defining the behavior of each component in the abstraction of the components surrounding it, and I would agree with any criticism given here. Thankfully, remedying this is as simple as moving an and gate from inside of the VHDL entity to the outside. My overall implementation of branch prediction also works correctly if the branch is predicted as not taken, despite my selected algorithm always predicting it as taken. If a more complex algorithm was implemented, the only necessary changes to the VHDL code would be to add some form of a writeback port to the branch prediction entity to update a branch history table. From there, new branch prediction algorithms could be implemented as a new VHDL

architecture of the currently existing branch prediction entity, requiring no further changes to the external system.

In any system that implements branch prediction, control hazards exist when the prediction algorithm is incorrect. To remedy this, flush all the subsequent instructions in the pipe whenever a branch is mispredicted. This solution is simple and effective but has one edge case in which the results are correct but beg to be improved. If the branch instruction is Instruction 0, and the branch is predicted as taken, what should be done with Instruction 1? Instruction 1 could get flushed, after all the branch was predicted as taken and I am not expecting to execute Instruction 1, but I must acknowledge that my branch prediction algorithm is not perfectly accurate. It is possible that this branch is not taken, and execution of the program must resume at Instruction 1. Effectively, this implementation would flush Instruction 1 in favor of inserting a nop, only to issue Instruction 1. The worst case is that Instruction 1 is an integer multiply instruction and the following instruction is dependent on the results of Instruction 1, causing the next instruction to stall until Instruction 1 is complete. The solution to this is straightforward: regardless of the branch prediction, if the branch instruction is Instruction 0, issue Instruction 1 as soon as possible and only flush Instruction 1 if the branch is not taken. If the branch is Instruction 1, never flush Instruction 0. Here is the second requirement for the hardware to know which instruction was the branch instruction, and the branch position signal is once again utilized. Unfortunately, in the edge case mentioned previously, it can not be guaranteed that Instruction 1 will be in Stage 1 of the even pipe, as Instruction 1 may have stalled due to a structural hazard or a data hazard. This possibility does not make the generation of flush signals straightforward.

The logic that generates the flush signals is contained in the component found in "flush_generator.vhd". The flushing logic for each pipe in the Instruction Decode and Register File stages is as follows: if the branch was Instruction 0, the branch was taken, and the instruction at this stage is Instruction 1, flush this instruction. Otherwise, flush this instruction if the branch was mispredicted. For Stage 1 of the even pipe, the flushing logic is just to flush the instruction if the branch was Instruction 0 and the branch was taken. There's no need to check whether this is Instruction 1 because it can only be either Instruction 1 or a nop. The only problem left to resolve is to determine how the flush generator can identify whether an instruction is Instruction 1 when the branch was Instruction 0. It is known that if the branch instruction was Instruction 0, the branch position signal will be 0. With that, we know that Instruction 1's program counter will be the branch instruction's program counter + 4, so each stage of the pipeline can be tested to see if Instruction 1 is found.

In the event of a branch misprediction, the correct branch target address computed in the branch pipe gets selected by the branch multiplexer in the Instruction Fetch stage. This value gets assigned to the instruction memory address and is used to compute the next program counter value.

```
architecture always_taken of branch_prediction is
begin
    predict_taken <= '1' and br_en;
    br_predict_address_comp: process(all)
    begin
        case br_mode is
            when '0' => target <= std_logic_vector(signed(pc) + resize(signed(i16 & "00"), 32));
            when '1' => target <= std_logic_vector(resize(signed(i16 & "00"), 32));
            when others =>  target <= (others => '0');
        end case;
    end process br_predict_address_comp;
end always_taken;
```

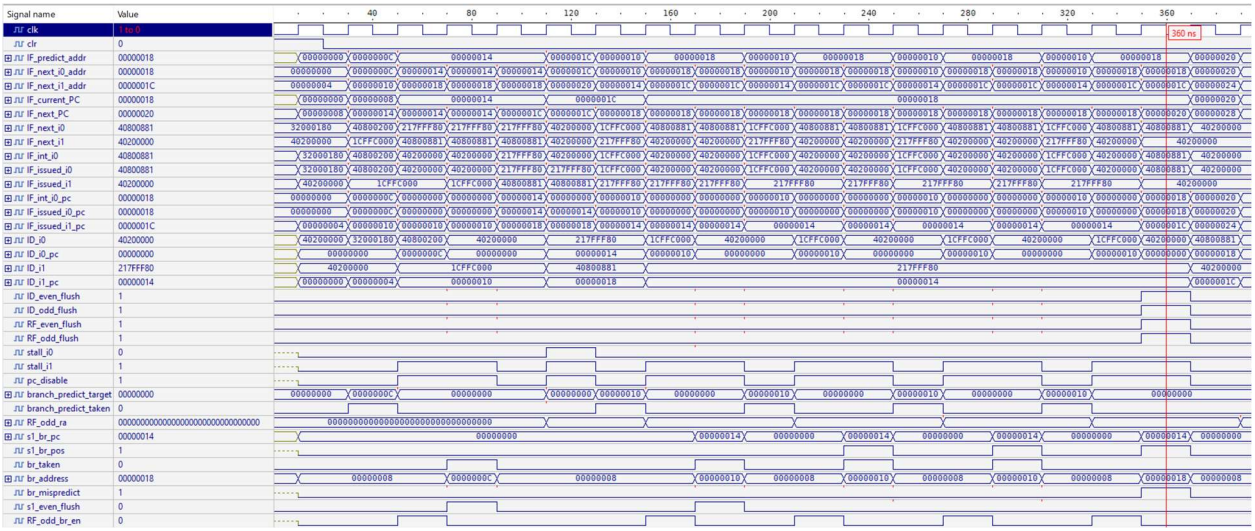
```
-- INSTRUCTION FETCH STAGE
pc : entity program_counter port map(
    clk => clk, clr => clr, pc_in => IF_next_PC, pc_out => IF_current_PC
);

pc_en_mux : entity mux_32 port map(
    a => std_logic_vector(signed(IF_next_i0_addr) + to_signed(8, 32)), b => IF_next_i0_addr, sel => (pc_disable and not br_mispredict), y => IF_next_PC
);

predict_mux : entity mux_32 port map(
    a => IF_current_PC, b => branch_predict_target, sel => branch_predict_taken, y => IF_predict_addr
);

branch_mux : entity mux_32 port map(
    a => IF_predict_addr, b => br_address, sel => br_mispredict, y => IF_next_i0_addr
);

IF_next_i1_addr <= std_logic_vector(signed(IF_next_i0_addr) + to_signed(4, 32));

IF_next_i0 <= instruction_memory(to_integer(unsigned(IF_next_i0_addr)));
IF_next_i1 <= instruction_memory(to_integer(unsigned(IF_next_i1_addr)));
```

Test Program

```
1  br 3
2  nop
3  nop
4  il r0, 4
5  ai r0, r0, -1
6  brnz r0, -1
7  il r1, 17
8  nop
9  nop
```

The first branch jumps to line four, which is located at address 0x0000000C. This loads a loop counter into r0, decrements it, and reaches a branch. The branch will be predicted taken, but the instruction at line 7 will still be issued, just in case the branch is not taken. When the branch is determined to be taken, this will be flushed. The branch will be taken until r0 reaches 0. When this happens, the branch will be predicted to be taken, but will predict incorrectly. The pipeline will need to be flushed and the program must resume execution at line 7, which is address 0x00000018. Nops will then be issued after this.

Output Waveforms



As seen, the first branch is predicted as taken, since branch_predict_taken is set. Two cycles later, the branch pipe determines that the branch is taken, and the instructions loaded into the decode and register file stages are not flushed. On the next clock cycle, the instruction to load the loop counter is issued, showing that the instruction at the predicted target is getting loaded properly. The branch is then successfully predicted and taken three times. On the fourth time the branch instruction is reached, the value in the preferred slot of r0 is 0, so the branch will not be taken. The branch prediction algorithm predicts that this branch is taken. Two cycles later, the branch pipe computes that this branch is not taken, sets the branch mispredict signal, and sets the signals to flush the instructions in the Instruction Decode and Register File stages. The next load immediate instruction after the branch is loaded, and nops get issued for all subsequent instructions because this program has completed.

Loading Instructions Generated By The Assembler

In the ESE545_CellSPU.vhd file, a process reads each line of a text file and fills in the instruction memory with each line it reads.

```
load_instructions : process
file input                : text;
variable L, reg_line      : line;
variable instruction      : std_logic_vector(0 to 31);    -- variable to read the instruction into.
variable instruction_num  : integer := 0;
variable errors           : integer := 0;
begin
    -- load instruction buffer contents
    FILE_OPEN(input, "instructions.txt", READ_MODE);
    while not endfile(input) and instruction_num < instruction_memory_size loop
        -- read the next instruction and put it in IB_instruction
        readline(input, L);
        read(L, instruction);
        instruction_memory(instruction_num) <= instruction;
        instruction_num := instruction_num + 4;
    end loop;

    while instruction_num < instruction_memory_size loop
        instruction_memory(instruction_num) <= (1 | 10 => '1', others => '0');
        instruction_num := instruction_num + 4;
    end loop;

    wait;
end process load_instructions;
```

The functionality of this process has been verified by the execution of every other program in this report, since this process is the one that was utilized to load the programs.

4x4 Integer Matrix Multiplication

I've included a file titled "4x4 Matrix Multiplication Instruction Layout.xlsx" that details the purpose of every instruction, shows how the instructions are issued into the even and odd pipes, and provides a list of registers and their associated usage. The associated assembly file is titled "matrix.s". I'll provide an explanation here that captures the approach I took with this task.

First, the program must load the matrices from the LS. This requires eight instructions that all use the odd pipe, which, if ordered sequentially in the program, would result in eight nops being inserted into the even pipe. If there's are any setup instructions that need to use the even pipe, this seems like a good place to insert them in the program.

I wrote out the equation for a 4x4 matrix multiplication on paper and determined that I'll need a mechanism to take one word from a register and place it in every word of a target register. I was naively going to resolve this using some sequence of selects, bit shifts, and or instructions, but I realized how inefficient this approach would be. I searched through the Cell SPU ISA to find a more optimal solution and realized the potential of the "Shuffle Bytes" instruction. I saw that I could generate address sequences in the rc register that would place the desired word from ra into the target register in a single instruction. I would simply need to load 32-bit immediate values into the four words of some general-purpose registers. This takes eight clock cycles and entirely uses the even pipe, so it can be done in parallel with the load instructions.

There are various ways to carry out the matrix multiplication. Since the rows are stored as four-element vectors in general-purpose registers, the most straightforward way to complete this matrix multiplication is to compute row 1 of the result matrix, then compute row 2, and so on. From a software perspective this is fine, but considering the implementation of the Cell SPU, this will result in numerous stalls as all the operations required to compute a row of the resulting matrix are dependent. An integer multiplication instruction takes seven cycles to complete, so each subsequent instruction would be stalled for six clock cycles. There's no way to overcome the data dependency, so if there's any independent work that can be done in these clock cycles, performance should improve significantly. I took some inspiration from the concept of vector processing. Even though all the instructions for a single row are dependent, there are four rows that need to be computed, and none of the operations across different rows are dependent. After an operation for row 1 is issued, an operation for row 2 can be issued while waiting for the previous operation to complete, and so on for rows 3 and 4.

While the multiplication and multiply-accumulate instructions are executing in the even pipe, the odd pipe is carrying out the shuffle byte instructions that broadcast the desired word from a row of matrix A to all four words of the target register to prepare for the next set of multiply-accumulate instructions.

Once all 16 multiply-accumulate instructions have been executed, the rows of the result matrix are stored back to the LS.

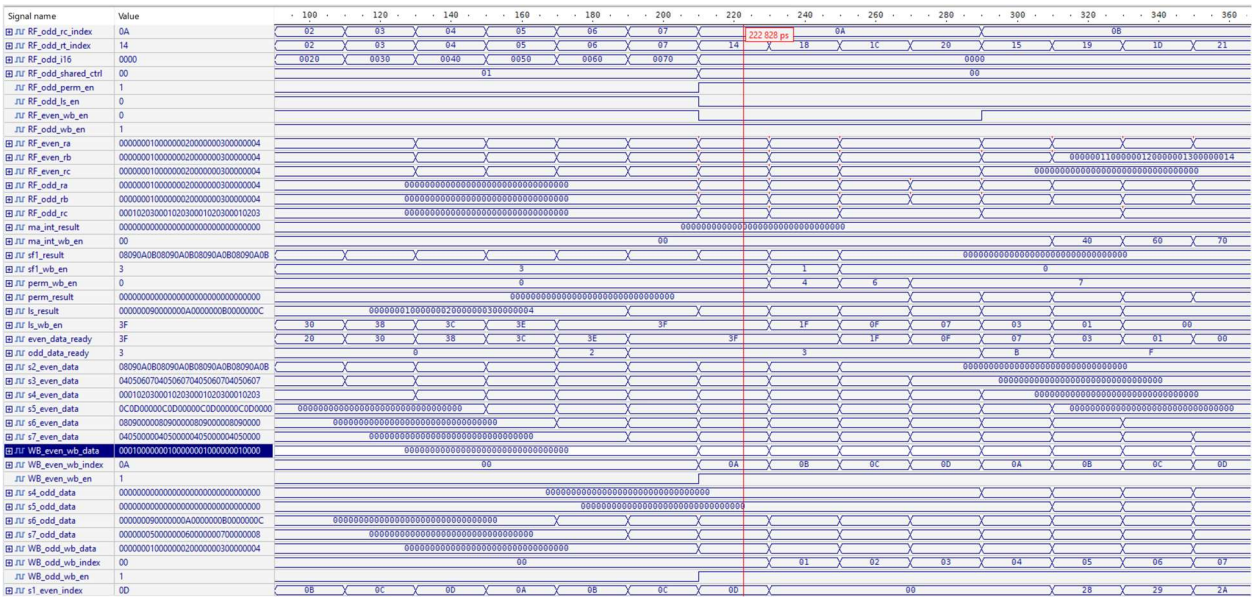
Test Input

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \begin{bmatrix} 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 \\ 25 & 26 & 27 & 28 \\ 29 & 30 & 31 & 32 \end{bmatrix} = \begin{bmatrix} 250 & 260 & 270 & 280 \\ 618 & 644 & 670 & 696 \\ 986 & 1028 & 1070 & 1112 \\ 1354 & 1412 & 1470 & 1528 \end{bmatrix}$$

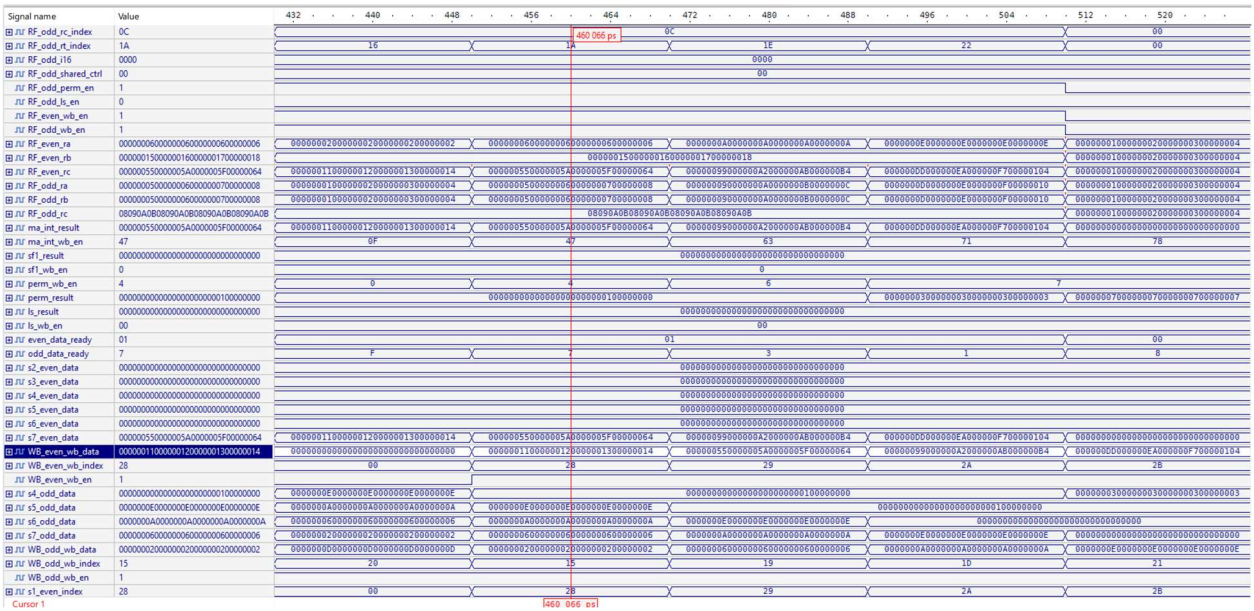
The load/store entity was modified to place these two arrays in row-major order starting at address 0 when the synchronous clear is set. The leftmost array is array A and starts at memory address 0. The array it is being multiplied with is array B and starts at memory address 64. The result array will be computed and stored in row-major order starting at memory address 128.

Output Waveforms

The complete output waveform has been submitted with the project HDL/Cell_SPU/Cell_SPU/src/4x4_Matrix_Waveforms. This report will include screenshotted highlights to display the functionality of each part of the program.

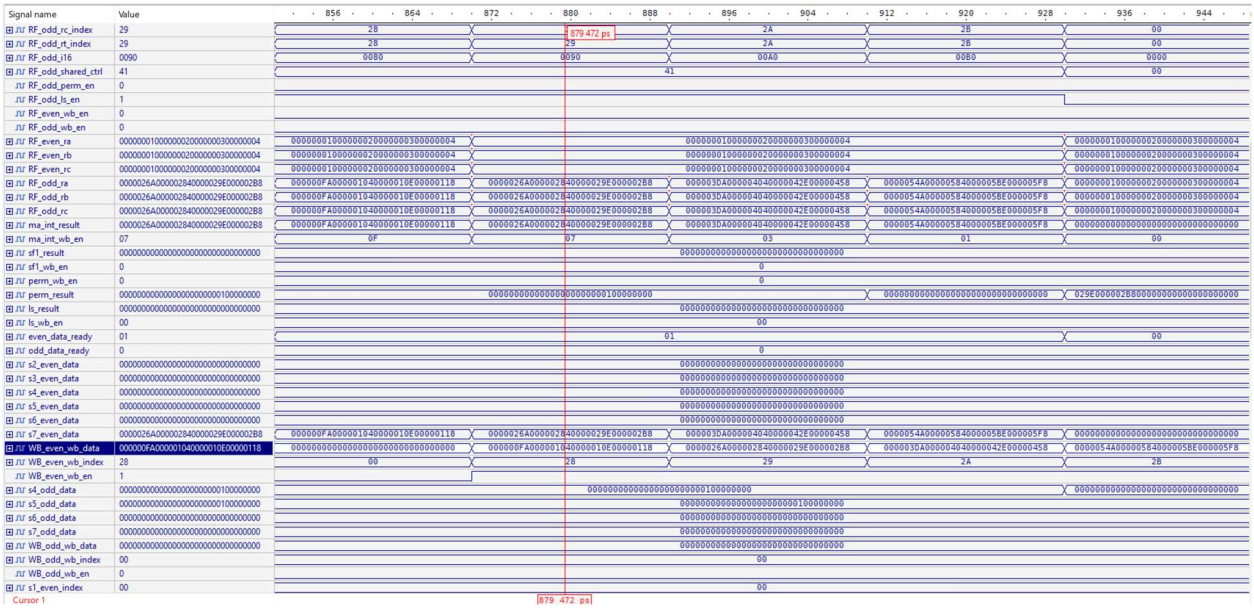


The above screenshot begins at 90ns and ends at 370ns. Within this screenshot, the propagation of the results of the load quadword, load immediate, and or immediate instructions can be seen. The cursor is located at the writeback stage of the first pair of instructions, in which row 1 of matrix A is being loaded into r0 and the immediate value to be used in the shuffle instructions is being loaded into the upper halfword or r10. Looking at the earlier stages of the pipeline, the results of the subsequent load quadword and load immediate instructions can be seen. Once the results hit the writeback stage, the writeback enable input is set, and these results get written back to the register file.



The above screenshot displays the waveforms from 430ns to 530ns. As seen towards the bottom of the image, the results of the multiply instructions are becoming available at Stage 7, propagating to the writeback stage, and setting the writeback enable signal. Once the data is

available at Stage 7, the next dependent instruction is at the register file stage, and the new result is forwarded to the RF_even_r[x] register signals found towards the top of the screenshot so that these instructions may begin execution. This same behavior occurs three more times, found over the intervals of 570ns to 670ns, 710ns to 810ns, and 850ns to 950ns. This last interval is of interest, as these should be the computed results for the matrix and will be displayed in the next screenshot.



The above screenshot shows the final results of the matrix multiplication. I've placed the values of this screenshot into matrix format below.

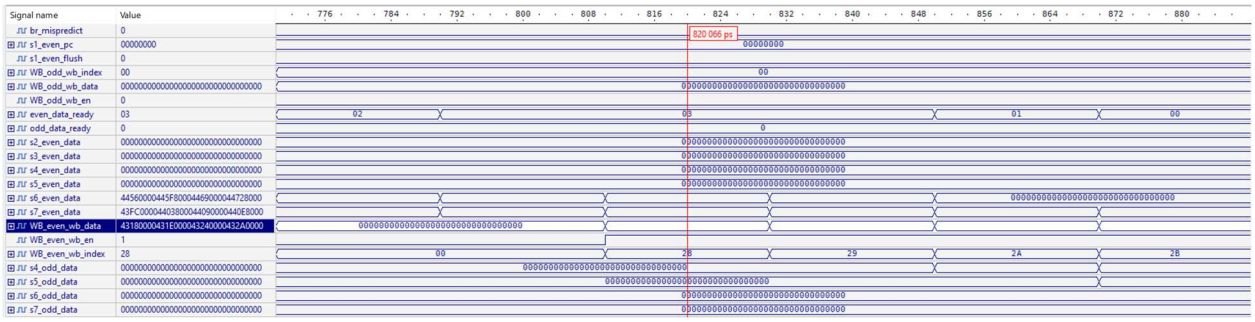
$$\begin{bmatrix} 0xFA & 0x104 & 0x10E & 0x118 \\ 0x26A & 0x284 & 0x29E & 0x2B8 \\ 0x3DA & 0x404 & 0x42E & 0x458 \\ 0x54A & 0x584 & 0x5BE & 0x5F8 \end{bmatrix} = \begin{bmatrix} 250 & 260 & 270 & 280 \\ 618 & 644 & 670 & 696 \\ 986 & 1028 & 1070 & 1112 \\ 1354 & 1412 & 1470 & 1528 \end{bmatrix}$$

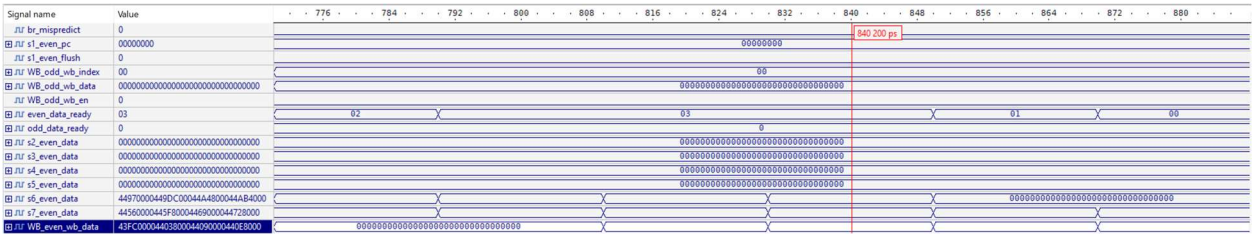
You may also observe that, once the final results are computed, the store instructions are issued, and the data is forwarded to the RF_odd_r[x] signals.

4x4 Single Precision Floating Point Matrix Multiplication

I've mistakenly gone into depth explaining the integer results when I now realize that the project requirement is 4x4 floating point matrix multiplication. The source code is included as "float_matrix.s", and the algorithm is identical to the integer floating point multiplication. The only difference between these is that the results for floating point operations are available at Stage 6 instead of Stage 7, so dependent instructions will be stalled for 1 less clock cycle.

For this test, I've loaded the first two matrices of the LS with values from 0.0 to 31.0, as suggested in the project summary table.





The final results can be found in the time interval 770ns to 890ns, as is shown in the two screenshots above. The data shown above is in row major order, with row 1 being computed first. This means the data can be shown in a matrix as is shown below.

$$\begin{bmatrix} 0x43180000 & 0x431E0000 & 0x43240000 & 0x432A0000 \\ 0x43FC0000 & 0x44038000 & 0x44090000 & 0x440E8000 \\ 0x44560000 & 0x445F8000 & 0x44690000 & 0x44728000 \\ 0x44970000 & 0x449DC000 & 0x44A48000 & 0x44AB4000 \end{bmatrix} = \begin{bmatrix} 152.0 & 158.0 & 164.0 & 170.0 \\ 504.0 & 526.0 & 548.0 & 570.0 \\ 856.0 & 894.0 & 932.0 & 970.0 \\ 1208.0 & 1262.0 & 1316.0 & 1370.0 \end{bmatrix}$$

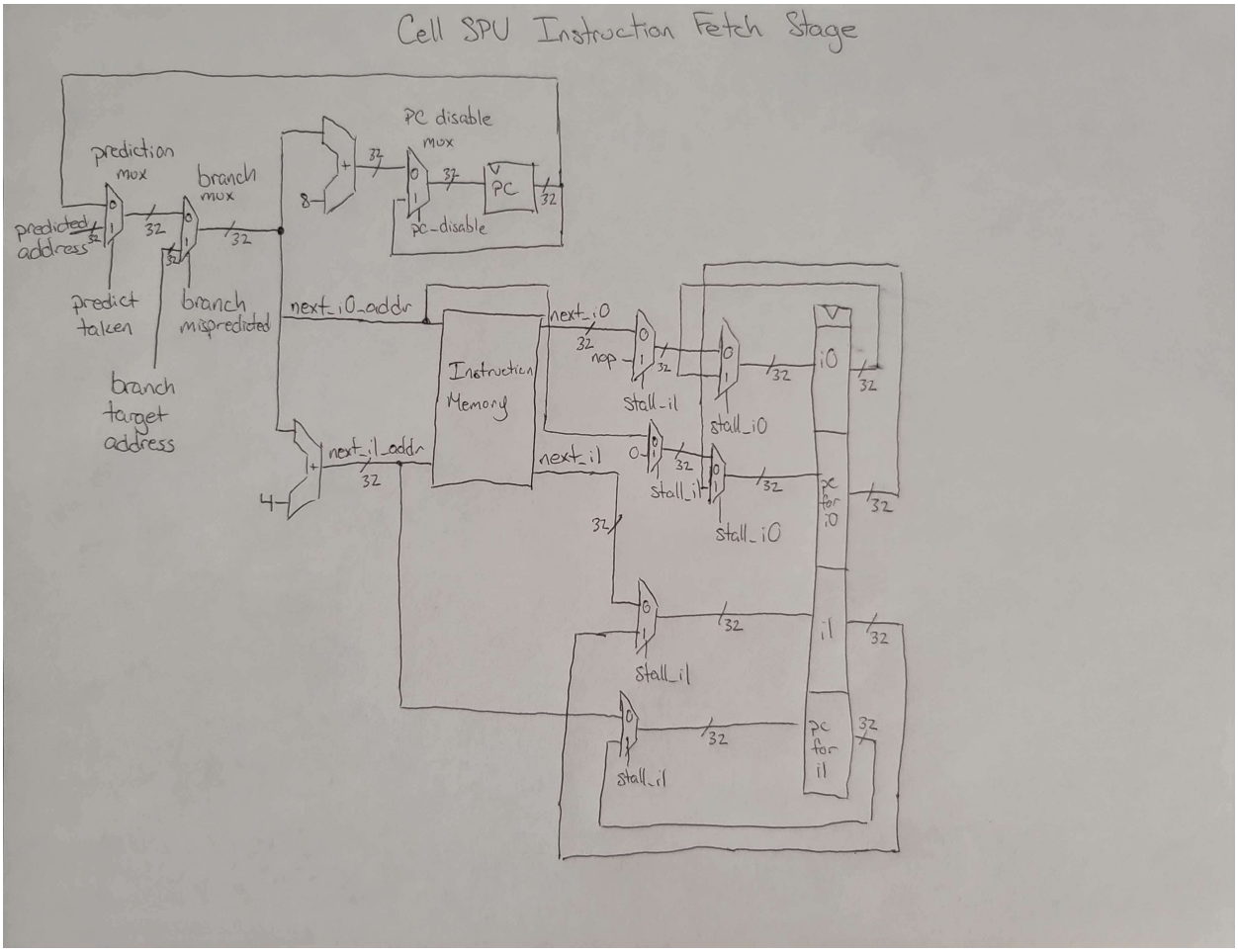
Assembler

The assembler is a simple text parser that reads the input one line at a time, and outputs the test vectors in ASCII format. The parser works by reading in the assembly instruction mnemonic and using this as a key to a hash map that stores the format of the instruction and the binary opcode for the instruction. Based on the format of the instruction, the parser enters a switch block that reads in the data in the expected format. Registers are labelled as r[n] for the nth register, from r0 to r127, and the conversion from a register mnemonic to a binary value uses another hash map. Found below are screenshots of the switch block, which is the primary block of logic.

```
while (!input.eof()) {
    std::string instr{};
    char operands[MAX_LEN];
    char* rt, * ra, * rb, * rc, * imm;
    // get first token
    input >> instr;

    std::pair<instruction_type, std::string> idk_what_to_name_this = instruction_hash.at(instr);
    instruction_type type = idk_what_to_name_this.first;
    std::string opcode = idk_what_to_name_this.second;

    switch (type) {
    case instruction_type::R:
        input.getline(operands, MAX_LEN);
        rt = strtok(operands, ", \t");
        ra = strtok(NULL, ", \t");
        output << opcode << binary_register_hash.at(std::string(ra)) << binary_register_hash.at(ra) << binary_register_hash.at(rt) << std::endl;
        break;
    case instruction_type::RR:
        input.getline(operands, MAX_LEN);
        rt = strtok(operands, ", \t");
        ra = strtok(NULL, ", \t");
        rb = strtok(NULL, ", \t");
        output << opcode << binary_register_hash.at(std::string(rb)) << binary_register_hash.at(std::string(ra))
            << binary_register_hash.at(std::string(rt)) << std::endl;
        break;
    case instruction_type::RRR:
        input.getline(operands, MAX_LEN);
        rt = strtok(operands, ", \t");
        ra = strtok(NULL, ", \t");
        rb = strtok(NULL, ", \t");
        rc = strtok(NULL, ", \t");
        output << opcode << binary_register_hash.at(std::string(rt)) << binary_register_hash.at(std::string(rb))
            << binary_register_hash.at(std::string(ra)) << binary_register_hash.at(std::string(rc)) << std::endl;
        break;
    case instruction_type::RI7:
        input.getline(operands, MAX_LEN);
        rt = strtok(operands, ", \t");
        ra = strtok(NULL, ", \t");
        imm = strtok(NULL, ", \t");
        output << opcode << itoab(atoi(imm), 7) << binary_register_hash.at(std::string(ra))
            << binary_register_hash.at(std::string(rt)) << std::endl;
        break;
```

```
case instruction_type::RI8:
    input.getline(operands, MAX_LEN);
    rt = strtok(operands, ", \t");
    ra = strtok(NULL, ", \t");
    imm = strtok(NULL, ", \t");
    output << opcode << itoab(atoi(imm), 8) << binary_register_hash.at(std::string(ra))
        << binary_register_hash.at(std::string(rt)) << std::endl;
    break;
case instruction_type::RI10:
    input.getline(operands, MAX_LEN);
    rt = strtok(operands, ", \t");
    ra = strtok(NULL, ", \t");
    imm = strtok(NULL, ", \t");
    output << opcode << itoab(atoi(imm), 10) << binary_register_hash.at(std::string(ra))
        << binary_register_hash.at(std::string(rt)) << std::endl;
    break;
case instruction_type::I16:
    input.getline(operands, MAX_LEN);
    imm = strtok(operands, ", \t");
    output << opcode << itoab(atoi(imm), 16) << "0000000" << std::endl;
    break;
case instruction_type::RI16:
    input.getline(operands, MAX_LEN);
    rt = strtok(operands, ", \t");
    imm = strtok(NULL, ", \t");
    output << opcode << itoab(atoi(imm), 16) << binary_register_hash.at(std::string(rt)) << std::endl;
    break;
case instruction_type::DFORM:   // i10 format, different assembly syntax
    input.getline(operands, MAX_LEN);
    rt = strtok(operands, ",() \t");
    imm = strtok(NULL, ",() \t");
    ra = strtok(NULL, ",() \t");
    output << opcode << itoab(atoi(imm), 10) << binary_register_hash.at(std::string(ra))
        << binary_register_hash.at(std::string(rt)) << std::endl;
    break;
case instruction_type::NOP:
case instruction_type::STOP:
    output << opcode << "00000000000000000000" << std::endl;
    break;
```

Instruction Fetch Diagram



One modification was made to this diagram after testing. It was observed that the stall signals could be set at the same time as the branch mispredict signal. If a branch is mispredicted, it is not desirable to load stalled instructions into the decode stage, so all the multiplexers that have pc_disable, stall_i0, or stall_i1 as a select input are now logically ANDed with the inverted branch mispredict signal to ensure that the instructions at the computed branch address are loaded.