Hazard516

ECE 449 16 Bit CPU

Department of Electrical and Computer Engineering

Michael Nicolaisen - V00979419

Mattias Kroeze - V00934043

Abstract

This document describes the design and implementation of this group's ECE449 project, being a 5-stage pipelined, 16-bit CPU, as this group dubs the HPU, or Hazard Processing Unit. Specifically, it describes how the HPU handles the execution of the given ISA, and how it handles hazards. After this, the report describes the results of this project, its achievements, and its downfalls.

Table of Contents

Introduction

This project involves designing and implementing a 5-stage pipelined 16-bit RISC-style processor on an FPGA. The provided Instruction Set Architecture (ISA) outlines three main instruction formats our CPU must be capable of executing to functionally run provided programs. Format A, B, and L instructions consist of arithmetic, branch, and memory operations. Our CPU implements two memory units: a dual-ported RAM module for data and instructions and a ROM unit containing a rudimentary BIOS used to communicate over the input and output ports with an STM32 companion board for downloading programs. Pipelined CPU architectures introduce significant efficiency improvements compared to sequential instruction execution, however, implementation requires significant planning and forethought due to the complexity and potential hazards introduced by the pipelined architecture.

Objective

This project aims to develop a functional 16-bit CPU capable of running programs written in the provided Instruction Set Architecture on FPGA hardware. This task requires a thorough and well-thought-out architecture to function. This includes designing a datapath, instruction decoder, ALU, branch logic, data and instruction memory, as well as a controller that can effectively monitor for and clear potential structural and data hazards. The final CPU should implement a 5-stage pipelined architecture. Pressing the “reset and load” button runs a BIOS off of the ROM module that communicates with an STM32 bootloader via the processor’s input/output. This will load a program from the STM32 into RAM. Pressing “reset and execute” will jump the program counter to the first instruction in RAM to execute the downloaded program.

CPU Architecture and Pipeline Design

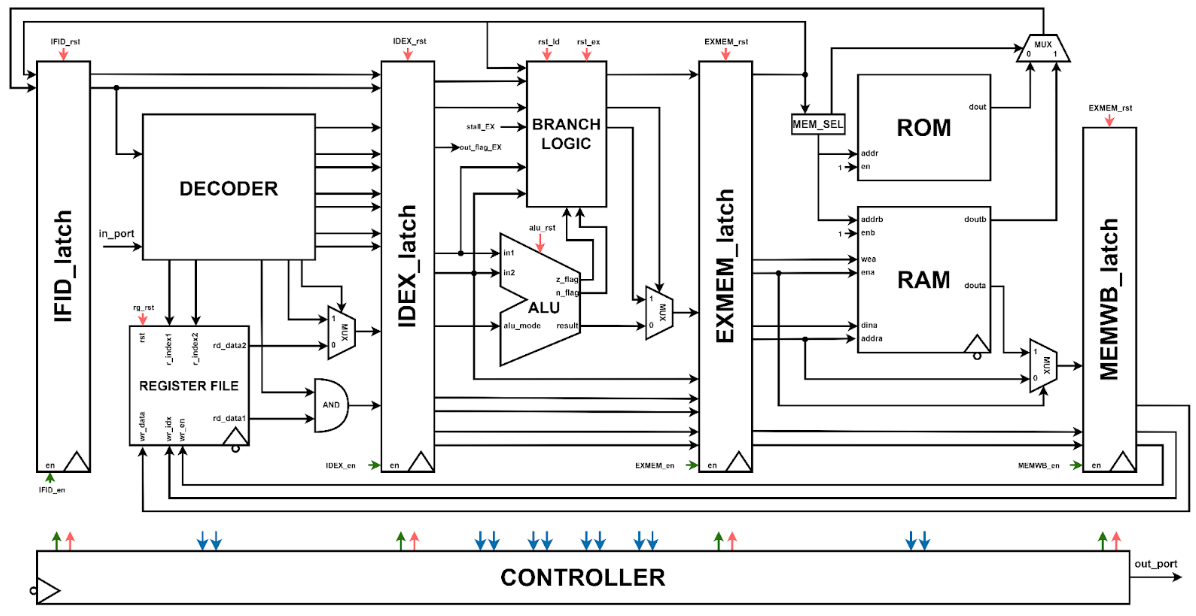


Figure 1: High-level CPU pipeline dataflow diagram.

The architecture of our 16-bit CPU follows the traditional 5-stage MIPS pipeline design, consisting of instruction fetch, decode, execution, memory, and writeback stages. *Figure 1* shows a high-level diagram of our dataflow, with individual signals left unlabeled to improve readability given the density within each stage. Each of these stages, along with our controller, will be outlined in detail throughout the rest of this section. The entire, unmodified architecture diagram can be viewed in higher detail in *Appendix A*.

To separate each stage, our architecture uses four separate latches: **IFID_latch**, **IDEX_latch**, **EXMEM_latch**, and **MEMWB_latch**. These latches are synchronous and update their outputs to match their corresponding input signals on the rising edge of the CPU clock. Additionally, each latch implements synchronous resets and enables, which are manipulated by the controller discussed at the end of this section. These latches are essential for holding stable values within each stage during program execution and for preventing race conditions. Specifically, **IFID_latch** holds the fetched instruction to be decoded, **IDEX_latch** stores decoded operands and control signals for execution, **EXMEM_latch** carries ALU results along with memory and writeback control signals to the memory stage and also latches the incremented PC, while **MEMWB_latch** holds the data to be written back to the register file.

Stage 1 - Instruction Fetch

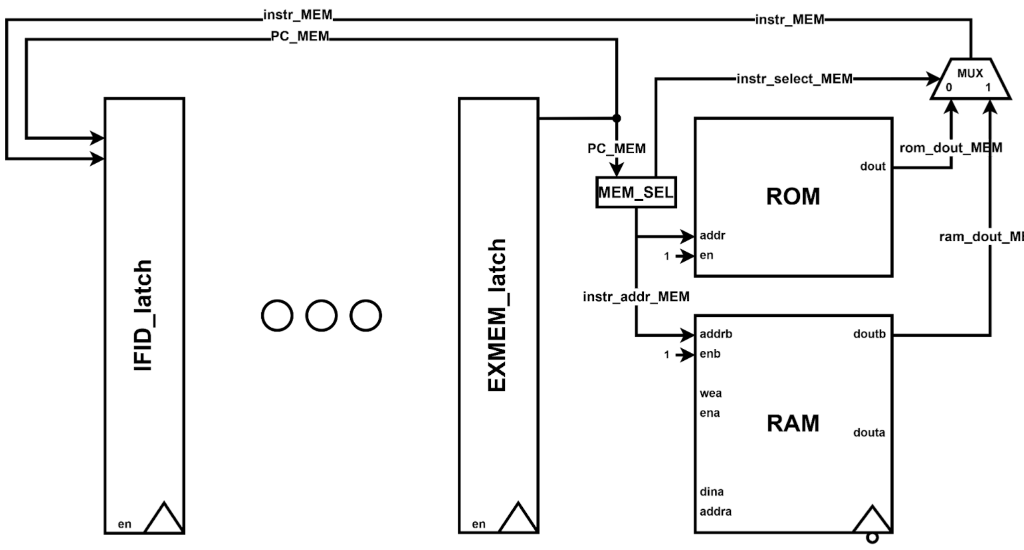


Figure 2: Instruction Fetch Pipeline Stage.

Our CPU uses two separate memory modules: a ROM for storing a rudimentary BIOS and a dual-ported RAM module for program instructions and data storage. Both ROM and RAM modules have a capacity of 1024 Bytes and perform asynchronous reads with the enable bit set persistently high. For our CPU to address the ROM and RAM modules separately, memory addresses 0 through 1023 are reserved for ROM, while addresses 1024 through 2047 are allocated for RAM. Therefore, a read from memory address 1024 should read from index 0 of the RAM module.

To perform this address conversion, we use the MEM_SEL module as shown to the left of the ROM module in *Figure 2*. MEM_SEL truncates **PC_MEM** to isolate the lower 10 bits such that a **PC_MEM** of 1028 is converted to an **instr_addr_MEM** value of 4, allowing for the correct addressing of RAM. However, both ROM and RAM will always read from **instr_addr_MEM** and return the contents of that memory address to their corresponding **dout_MEM** signals. To select between these two results, the MEM_SEL unit isolates bit 11 of **PC_MEM** and passes it to a MUX such that the result from ROM is selected if the PC is below 1024, and RAM will be selected if it is greater (bit 11 is set).

The resulting MUXed instruction in **instr_MEM,** as well as the corresponding program counter in **PC_MEM,** are routed to the IFID latch to be decoded in the following clock cycle.

Stage 2 - Instruction Decode

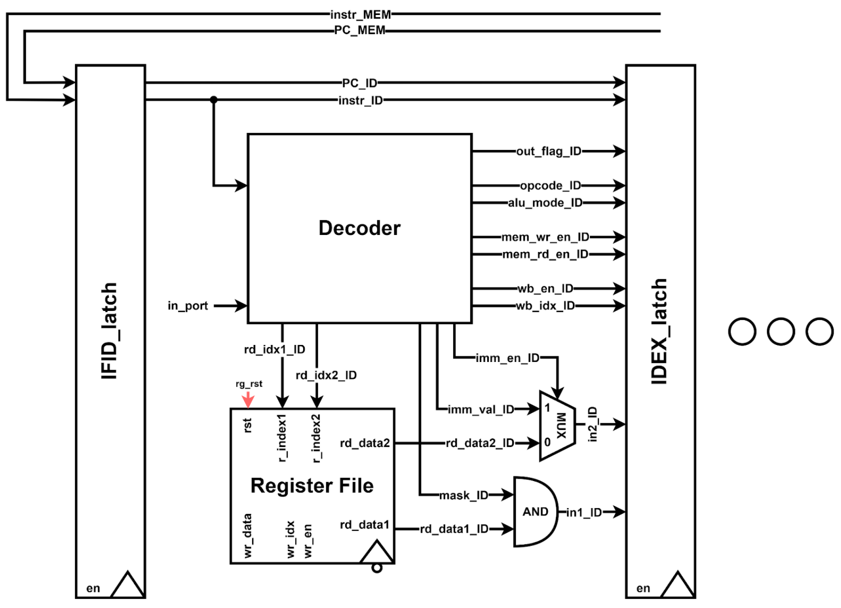


Figure 3: Instruction Decode Pipeline Stage.

Our instruction decoder stage latches incoming instructions from the fetch stage and extracts bits [15:9], which contain the instruction's opcode. This opcode is used in a case statement to determine the instruction format, operand sources, and required control signals. The Instruction Set Architecture (ISA) defines three major instruction formats: Format A, B, and L, which correspond to arithmetic, branch, and memory operations, respectively.

The **opcode_ID** signal specifies the instruction type for the Branch Logic unit, while **alu_mode_ID** defines the ALU operation to be performed. If the instruction requires a result to be written back to a register, **wb_en_ID** is asserted, and **wb_idx_ID** specifies the destination register. Memory operations are flagged by asserting **mem_rd_en_ID** or **mem_wr_en_ID** for reads and writes, respectively.

Instructions that read from the register file will specify source addresses using **rd_idx1_ID** and **rd_idx2_ID,** and the results will be available at **rd_data1_ID** and **rd_data2_ID**. The values in **rd_data1_ID** can be conditionally masked using a bitwise AND with **mask_ID**, which is useful for isolating the upper or lower byte in *LOADIMM* instructions or for forcing **in1_ID** to zero if required. By default, **mask_ID** is set to 0xFFFF, applying no masking.

For instructions involving immediate values, the decoder will assert **imm_en_ID** to MUX **in2_ID** to the immediate value provided by **imm_val_ID**, overriding the default value read from the register file at address **rd_idx2_ID**.

For an *IN* instruction, the decoder reads the value from the **in_port** signal and passes it as an immediate value, **wb_en_ID** is asserted, and **wb_idx_ID** specifies the destination register address.

For writing to the output port, *OUT* instructions assert **out_flag_ID** and read the source register via **rd_idx1_ID.** In the execution stage, the controller monitors the value of **out_flag_EX;** if asserted, the controller will latch the value in **in1_EX** to the processor's **out_port**.

Stage 3 - Execution

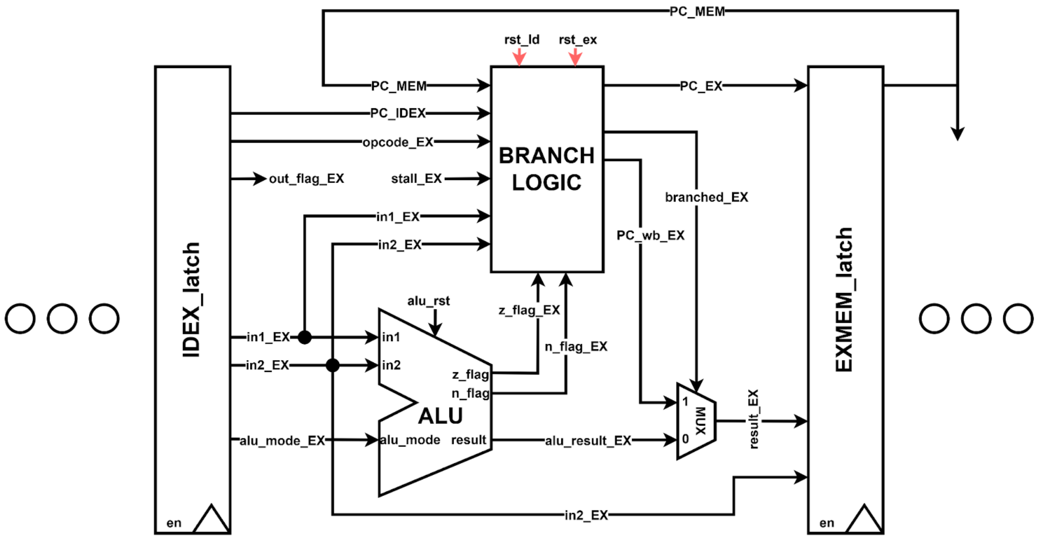


Figure 4: Execution Pipeline Stage.

ALU operations specified by the **alu_mode_EX** signal drive the ALU to perform arithmetic operations on the contents of the **in1_EX** and **in2_EX** signals, outputting the result to **alu_result_EX**. For *TEST* operations, the ALU asserts **z_flag_EX** and **n_flag_EX** if the value of **in1_EX** is zero or negative, respectively. When the CPU performs memory operations, **alu_mode_EX** is set to no op, allowing us to pass the memory address through the ALU from **in1_EX** to **alu_result_EX**. If writing to memory, **in2_EX** contains the data to write and is passed directly to the EXMEM_latch.

The Branch Logic unit monitors the **opcode_EX** signal to correctly manipulate the program counter, depending on the operands of the instruction in the execution phase. For non-branch operations, the Branch Logic unit sets **PC_EX**, the program counter for the next fetched instruction, to **PC_MEM** + 2 to point to the next instruction word in memory. If a branch operation is specified by **opcode_EX**, the Branch Logic unit checks the values of **z_flag_EX** and **n_flag_EX** to determine whether a conditional branch should be taken. Branch offsets are decoded as immediate values and read from the **in2_EX** signal. For relative branches, **PC_IDEX** contains the address of the currently executing instruction, and the resulting program counter **PC_EX** is calculated as **PC_IDEX** + 2 * **in2_EX**. For absolute branches, the value in register **ra** is read into **in1_EX**, and **PC_EX** becomes **in1_EX** + 2 * **in2_EX**.

The signals **branched_EX** and **PC_wb_EX** indicate whether a branch was taken and the value of the program counter before the branch, respectively. For *BR.SUB* operations, **wb_en_EX** must be asserted, and **wb_idx_EX** must address **r7**. In this case, **PC_EX** is set to the contents of **in1_EX** and **branched_EX** MUXes the value of **result_EX** to **PC_wb_EX** rather than the default **alu_result_EX**, and is, as a result, written back to **r7.** When a *RETURN* operation is executed, **PC_EX** is vectored back to the program counter before the branch, stored in **r7**.

The **stall_EX** flag is asserted by the controller if a RAW hazard is detected and will prevent the Branch Logic unit from incrementing the program counter. When signals **rst_ld** and **rst_ex** are asserted, the program counter is vectored to the values 0x0002 and 0x0000, respectively. This is imperative for user interaction with the BIOS code within the ROM.

Stage 4 - Memory Operations

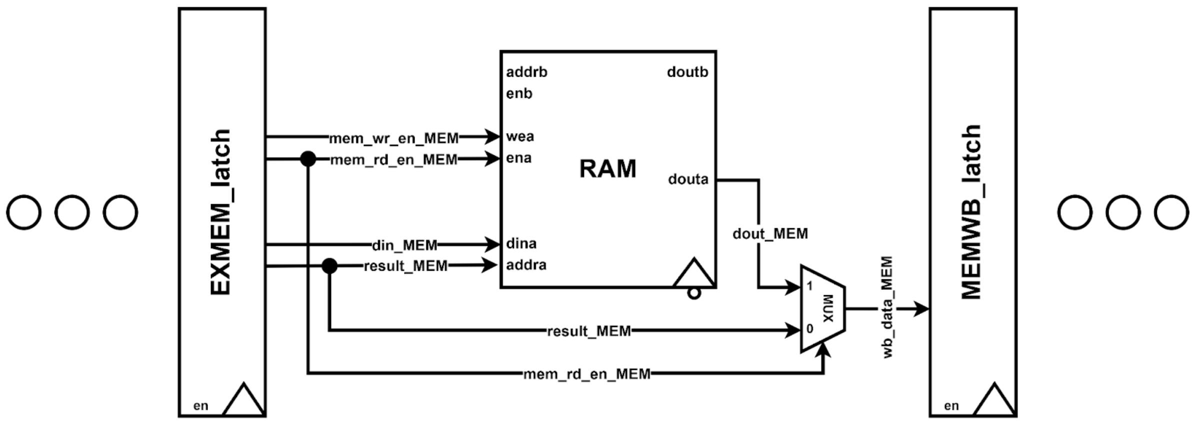


Figure 5: Execution Pipeline Stage.

Within the instruction decoder stage of our pipeline, if a memory operation is detected, the **mem_wr_en_ID** and **mem_rd_en_ID** control signals are asserted for memory write and read operations, respectively.

In the memory stage of the pipeline, if the instruction does not require memory access, both **mem_wr_en_MEM** and **mem_rd_en_MEM** will be deasserted, ensuring the ALU result in **result_MEM** is passed directly to the writeback path.

However, if the instruction requires a memory read, the decoder will have asserted **mem_rd_en_ID** and **wb_en_ID,** and specified **wb_idx_ID**, as to write back the read value to the specified register. A read on RAM port A is enabled by **mem_rd_en_MEM,** and douta will return the value stored at the address specified by **result_MEM**, which was passed through the ALU. The **mem_rd_en_MEM** control signal MUXes **wb_data_MEM** to select the read value from **dout_MEM** instead of the default ALU result.

Alternatively, if the instruction performs a memory write, **mem_wr_en_MEM** will be asserted, **result_MEM** will specify the address, and **din_MEM** will contain the value to write, passed through the execution stage in signal **in2_EX**.

Stage 5 - Writeback

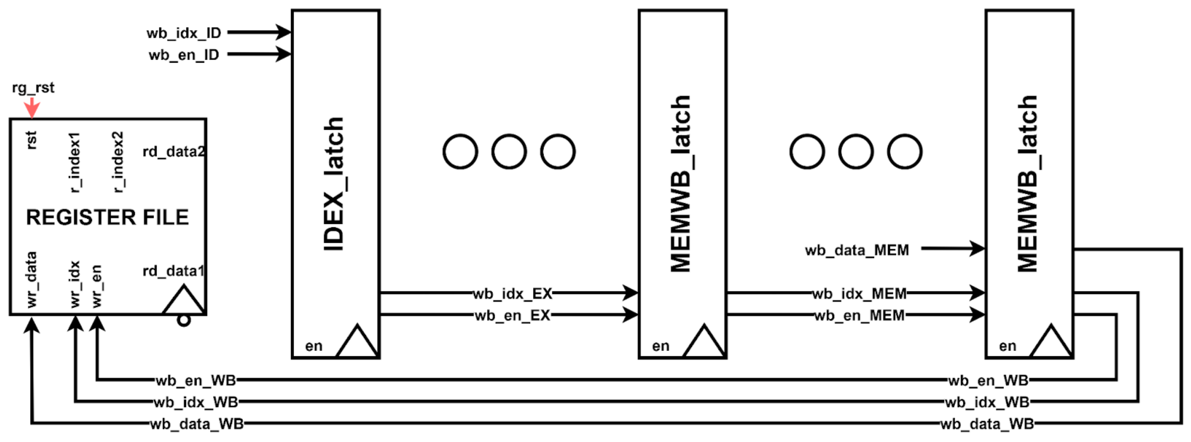


Figure 6: Execution Pipeline Stage.

If the result of an instruction requires to be written back to a register, the decoder would assert **wb_en_ID** and specify the destination with **wb_idx_ID**. These values are latched and would travel along with the instruction operands as they pass through our pipeline. On the rising clock edge, the MEMWB latch will latch the final **wb_en_WB, wb_idx_WB,** and **wb_data_WB** values. At the following falling clock edge, our register file will write the contents of **wb_data_WB** to the register specified by **wb_idx_WB** if the **wb_en_WB** signal is asserted.

Controller - Hazard Management

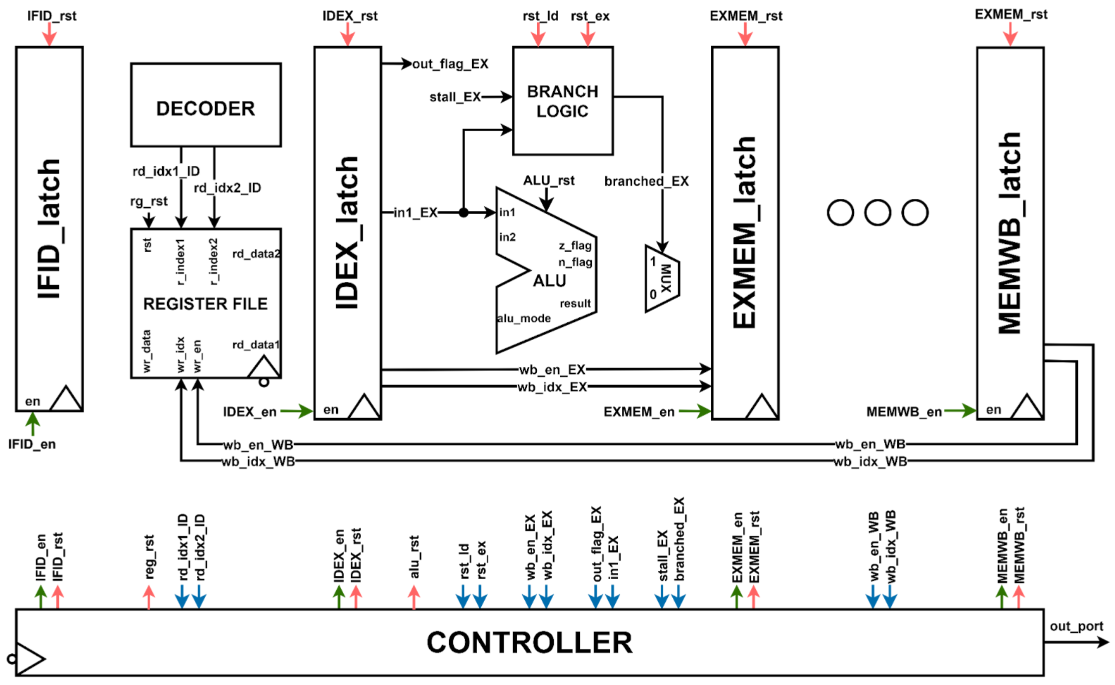


Figure 7: Controller Relevant Signals and Modules

Due to the added complexity of a pipelined implementation, the controller must detect and handle two major types of hazards. The first is a Read After Write (RAW) hazard, which occurs when an instruction in the decode stage attempts to read a register that is still waiting to be updated by a previous instruction that has not yet reached the writeback stage. Without proper handling, this would result in the instruction reading a stale value, leading to flawed program execution.

To address this, the controller monitors signals across all pipeline stages and implements an 8-bit pending_wb register, which tracks whether a register has an incoming writeback. Once an instruction passes the IDEX latch, it is considered to have begun execution. If the controller observes **wb_en_EX** asserted, the bit in pending_wb corresponding to **wb_idx_EX** is set. Conversely, in the writeback stage, if **wb_en_WB** is asserted, indicating a value is being written to the register file, the corresponding bit for **wb_idx_WB** is cleared.

Within the decode stage, the controller checks the source registers specified by **rd_idx1_ID** and **rd_idx2_ID**. If either of these addresses has a corresponding bit set in pending_wb, the instruction must stall until the writeback completes and the bit is cleared. In this case, the controller asserts **stall_EX**, preventing the program counter from incrementing, and de-asserts **IFID_en**, holding the current instruction in the decode stage. It also asserts **IDEX_rst**, which flushes the execution stage on the next rising clock and effectively inserts a No-op, allowing the pipeline to continue moving without extending the execution of a single instruction over multiple clock cycles. Once the writeback clears the corresponding bit in pending_wb, the controller de-asserts **stall_EX** and **IDEX_rst** and re-enables **IFID_en**, allowing the pipeline to resume normal execution.

The Branch Logic unit implements no branch prediction behavior; even in the case of an unconditional branch, we assume the branch is not taken and fetch the following instruction at the program counter + 2. This introduces a control hazard as if the branch is taken in the exection stage, the following instructions are already in the fetch and decoder stages and will break program flow if allowed to execute. Therefore, our controller monitors the **branched_EX** signal from our Branch Logic unit. If the signal is asserted, indicating a branch was taken, then we assert both **IFID_rst** and **IDEX_rst**, such that in the following clock cycle, the fetched instruction will be at the correct program counter, and the invalid instructions are replaced with No-ops are in both the decoder and execution stages, while the branch and prior instructions propagate through the rest of our pipeline. This effectively flushes our pipeline, ensuring no control hazards.

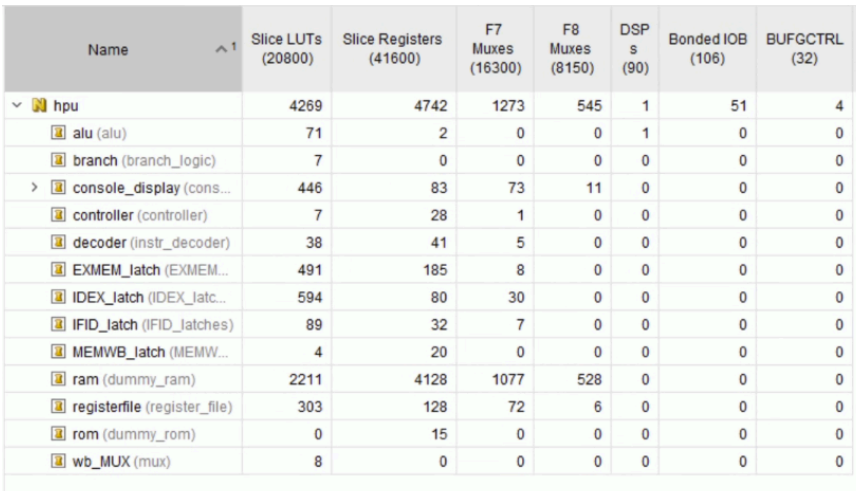
Additionally, when the **reset_ld** and **reset_ex** signals are asserted, the controller will assert signals **IFID_rst**, **IDEX_rst**, **EXMEM_rst**, **MEMWB_rst**, **reg_rst**, and **alu_rst**, effectively reinitializing our pipeline and registers to default values. Our controller performs all actions on the falling clock edge. This ensures that all values read by the controller have already been latched and that any actions taken will be applied in the following rising edge.

Results

In this section, we compare two metrics describing the Hazard516 core, being the frequency or max clock rate, and the hardware utilization

Hardware Utilization

Firstly, to evaluate our system, we ran the synthesizer and checked the hardware utilization of our system. The figure below shows the result of this procedure:



Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	F8 Muxes (8150)	DSPs (90)	Bonded IOB (106)	BUFGCTRL (32)
hpu	4269	4742	1273	545	1	51	4
alu (alu)	71	2	0	0	1	0	0
branch (branch_logic)	7	0	0	0	0	0	0
console_display (cons...	446	83	73	11	0	0	0
controller (controller)	7	28	1	0	0	0	0
decoder (instr_decoder)	38	41	5	0	0	0	0
EXMEM_latch (EXMEM...	491	185	8	0	0	0	0
IDEX_latch (IDEX_latc...	594	80	30	0	0	0	0
IFID_latch (IFID_latches)	89	32	7	0	0	0	0
MEMWB_latch (MEMW...	4	20	0	0	0	0	0
ram (dummy_ram)	2211	4128	1077	528	0	0	0
registerfile (register_file)	303	128	72	6	0	0	0
rom (dummy_rom)	0	15	0	0	0	0	0
wb_MUX (mux)	8	0	0	0	0	0	0

Figure 8: List of hardware components and their utilization

To further understand the hardware utilization of our system, we put together the following bar chart, where the utilization is shown in terms of percentages:

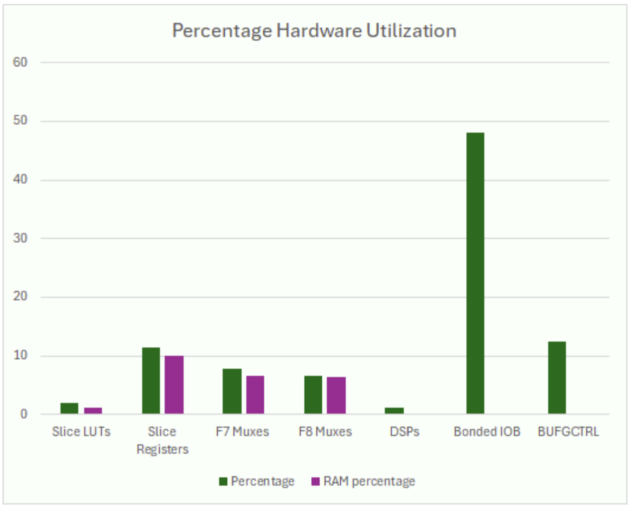
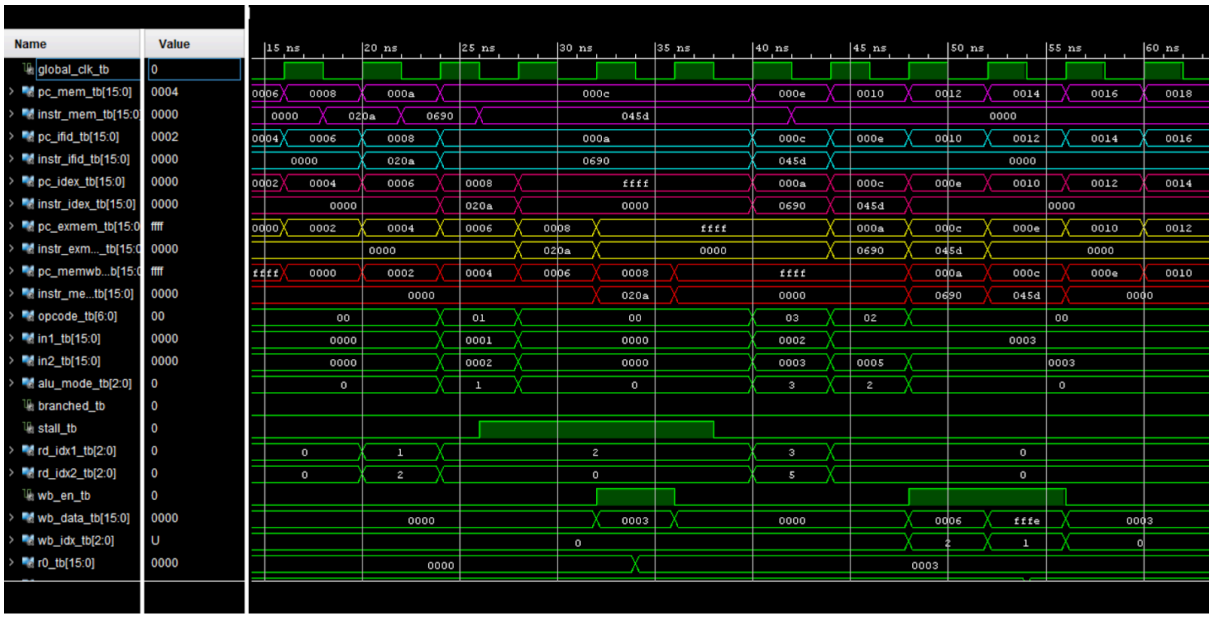


Figure 9: Percentage Utilization of RAM and HPU system (HPU includes RAM).

Simulation Screenshots

In this section, we will showcase the simulation screenshots demonstrating the core functionalities of the HPU. Note that all of these screenshots are available in a larger, rotated format in appendix B.

Format A



Name	Value
global_clk_tb	0
pc_mem_tb[15:0]	0004
instr_mem_tb[15:0]	0000
pc_ifid_tb[15:0]	0002
instr_ifid_tb[15:0]	0000
pc_idex_tb[15:0]	0000
instr_idex_tb[15:0]	0000
pc_exmem_tb[15:0]	ffff
instr_exm..._tb[15:0]	0000
pc_memwb...b[15:0]	ffff
instr_me...tb[15:0]	0000
opcode_tb[6:0]	00
in1_tb[15:0]	0000
in2_tb[15:0]	0000
alu_mode_tb[2:0]	0
branched_tb	0
stall_tb	0
rd_idx1_tb[2:0]	0
rd_idx2_tb[2:0]	0
wb_en_tb	0
wb_data_tb[15:0]	0000
wb_idx_tb[2:0]	U
r0_tb[15:0]	0000

Figure 10: Format A timing diagram

The timing diagram above shows the execution of the following instructions:

PC	**Instruction**
	ADD R0 R1 R2
0x0008	0x020A: 0000001 000 001 010
	MUL R2 R2 R0
0x000A	0x0690 : 0000011 010 010 000
	SUB R1 R3 R
0x000C	0x045D : 0000010 001 011 101

1. The first incoming instruction shown is 020a, which translates to ADD R0, R1, R2. This instruction appears at approximately 17.5 ns in the simulation.
2. At the 24ns mark at the **in1_tb** and **in2_tb** signals, you can see that **in1_tb** and **in2_tb** go to 1 and 2, respectively. These signals correspond to the inputs of the ALU in the execution stage of our pipeline. These signals are the latched result of the two register file read operations at the decode stage of the pipeline, with R1 containing the integer 1 and R2 containing the integer 2 for demonstration purposes.
3. Simultaneously, we can see that **alu_mode_tb** changes to 1, which puts the ALU in the addition mode, computing the addition of in1 and in2. Additionally, when the instruction is at this stage (R0 <- R1 + R2), the next instruction is being latched into the IFID latch, which happens to be MUL R2, R2, R0. Because this next instruction requires the contents of R0, which has not been written to yet, the controller unit recognizes this and sets the stall flag high, which one can see at the signal **stall_tb**, which stalls the pipeline until R0 has been written to.
4. The output of the ALU is asynchronous, processed, and written to the EXMEM latched, which brings the operation to the memory stage. Here, at 32ns, one can see that the **wb_en_tb** signal goes high. This is the register write back signal at the memory stage. We can see here that the write back was correctly asserted as per the requirements of the instruction.
5. Additionally, at the same time, one can see that the **wb_data_tb** line turns to the value 3, which shows the ALU correctly computed the addition of 1 + 2, and that this signal is about to be written back to R0. If we had not chosen R0 as the writeback register, one would see the signal **wb_idx_tb** change to the address of the register to be written back to (in this case, it's R0). At the end, these three signals, **wb_en**, **wb_data**, and **wb_idx**, are passed through to the MEMWB latch, which when clocked, brings them to the write back stage.
6. At the writeback stage, the three signals previously mentioned are presented to the write ports of the register file and are written to the register file on the following falling clock edge. In this case, one can see at the bottom of the timing diagram that the register R0 now contains the value 3.

Format B

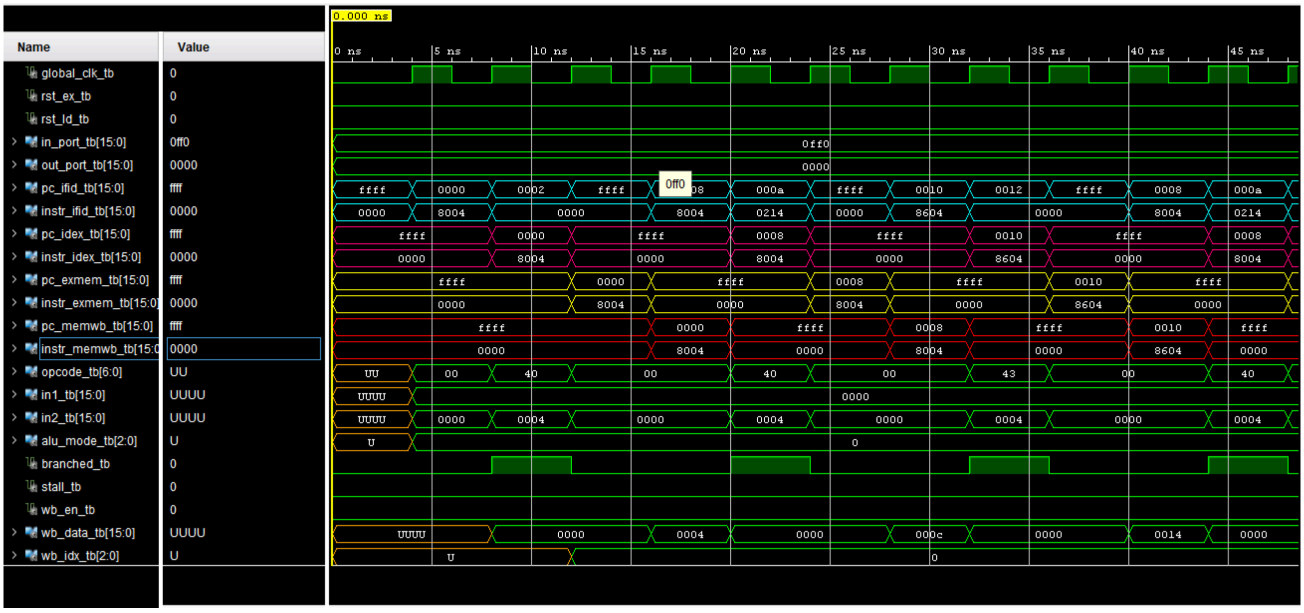


Figure 11: Format B timing diagram

Firstly, note that the instructions at play here are:

PC	Instruction
0x0000	BRR 4 0x8004: 1000000 000000100
0x0008	ADD R0 R2 R4 0x0214 : 0000001 000 010 100
0x000A	BR 4 0x8604 : 1000011 000000100

1. At the start of the timing diagram, one can see by inspecting the **instr_ifid_tb** signal that the first instruction to appear is the 0x8004 instruction, which is BR + 4. At this point, the instruction is at the decode stage, in which the opcode is sent directly to the IDEX latch, to be ready for the execution stage. Additionally, the immediate value of + 4 is passed through the immediate value MUX (at the decode stage), with the immediate value enable signal going high. In this way, the pipeline chooses to pass through the immediate value given by the instruction rather than the **rd_data2** of the register file. This immediate value is, like the opcode, given to the IDEX latch to be used by the execution stage.
2. In the next stage, being the execution stage, the immediate value along with the opcode, and the program counter are given to the **branch_logic** block, which executes both increases the program counter by twice the specified amount (to preserve the byte addressable nature of the memory unit), sets the branched signal

high. The **branched_tb** signal can be seen under **alu_mode_tb** signal, and one can see that it is raised high at 7ns due to this instruction. At the same time that this instruction is at the execute stage, the instruction at 0x0002 is being decoded. In this case, the instruction at 0x0002 is a NOP.

3. The next stage is the memory stage, in which the next instruction is prepared to be brought into the decoder. With the new program counter, the memory is ready from 0x0008, where our next instruction is. However, at this stage, the instruction at 0x0002 is being executed. To avoid this instruction making any unintended memory writes, we have to “flush the pipeline”. In which, we turn the program counter to 0xFFFF, reset the IDEX latch, and pass NOPs through. In this way, the instruction at 0x0002 is “thrown out”. If one looks at the **pc_ifid_tb** signal, starting at 12ns, they would see the program counter turn into 0xFFFF, and then turn to 0x0008, which indicates that the next instruction, already at the execute stage, is from 0x0008.
4. The instruction at 0x0008, being ADD R0, R2,R,4 is written back, which can be seen at 16ns, where the **wb_data_tb** signal changes to 0x0004, writing back the number 4 to R0.
5. In the next stage, at the same time that the instruction at 0x008 is being executed, the instruction at 0x000A, which happens to be BR 4, is being decoded.
6. At the next stage, BR 4 is in execution, where the branch logic unit changes the program counter to 0x0008 (double the immediate value, 4), and the pipeline is flushed to get rid of the instruction at 0x000C.
7. After this, an infinite loop occurs as the pipeline bounces between the instructions at 0x008 and 0x000A, which can be seen as the **branched_tb** periodically goes high.

Format L

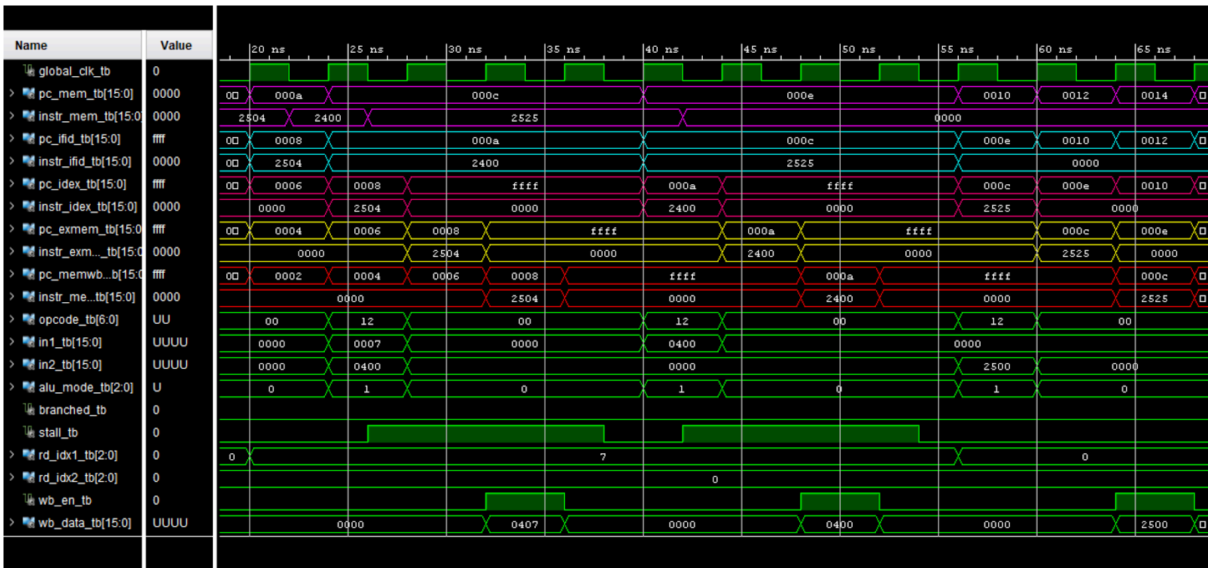


Figure 12: Format L timing diagram

Note that the figure above shows the execution of the program shown below:

PC	**Instruction**
0x0008	Loadimm Upper 4 0x2504: 0010010 1 00000100
0x000A	Loadimm Lower 0 0x2400 : 0010010 0 00000000
0x000C	Loadimm Upper 37 0x2525 : 0010010 1 00100101

1. Firstly, the program starts at 0x008 when the instruction 0x2504, which happens to be Loadimm.Upper 4, is loaded into the IFID latch, where, in a clock cycle, it is decoded. One can see the instruction loaded in at the blue signals at the top of the timing diagram, specifically at 20ns. In the decoding stage, the immediate value is prepared for the execution and, eventually, the write back stage, on top of which, the pending write back flag is asserted for R7 (as seen on signal **pending_wb_tb [7]** at 22ns). Thus, in this stage, the contents of R7 are read, and its upper bits are masked to 0. Since R7 initially contained the value 0x0007, then became 0x0007 (didn't change as the upper 8 bits are already 0x00. Additionally in this stage, the immediate value flag is set high, which lets the immediate value of 0x0400 pass through to the IDEX latch, ready for the next stage. The last thing to consider, is that in this stage, the decode asserts the ALU mode to be "001", which in the next stage will set the ALU into the "addition" mode.
2. In the next stage, the ALU is set to add the two numbers at its input, being 0x0400 and 0x0007, which can be seen at the two signals **in1_tb**, and **in2_tb**, which change to their respective values at 24ns. Additionally, when the first instruction being

Loadimm.Upper 4 is being executed, the next instruction being Loadimm.Lower 0 is being decoded. At this time, a RAW error is detected, as the second instruction requires the contents of R7 while its pending-write back flag is asserted. To address this, the controller sets the stall flag high (as seen at 25ns), which halts the decoder until R7 is written to, and its pending-write-back flag is lowered. Only after this, can the pipeline proceed.
3. At the memory stage, the write back index of 7, the write-back data of 0x0407, and the **wb_en** line is passed through to the MEMWB latch.
4. At the write-back stage, the data of 0x0407 is written to R7, and the **pending_wb** flag is set low, but then set high again as the next instruction, Loadimm.Lower 0, also writes back to R7.

Discussion

The Hazard516 processor lacks at least 2 basic features, firstly being the ability to execute push and pop instructions, and secondly, the ability to perform branch prediction. Push and pop instructions, and by extension, the stack, is an important part of a CPU, as it allows for subroutines and other programmatic structures to be written easily, as manual context switching would not be needed, and would allow for easier storage of local variables in memory. For this reason, if given more time, once we would have confirmed a working minimum viable product, the push and pop instructions would have been our first or second point of improvement.

Branch prediction would also be important for us to implement once the minimum viable product is working. This is because even a simple branch prediction algorithm such as a pattern history table can greatly reduce the number of stalls our system would have to undergo, which would increase the overall CPI of our system. This would, unfortunately, require a complete re-working of out stall and flush systems, which would take a considerable amount of effort.

One feature that the Hazard516 does implement well is the automatic RAW hazard control, in which write-back pending flags are raised for each register in the register file, and if an instruction needs to read from a register that hasn’t been written to yet, the system will stall until the register has been written to. This allows the programmer to not have to conditionally insert NOPs to avoid these hazards.

Throughout the design and testing process, we found more than a few errors that had to be overcome to move towards a working solution. Below, we will highlight three of the most significant challenges we faced.

Firstly, we had the following problem with the stall logic:
Consider the following program:

1. Loadimm.Upper 0x50
2. Loadimm.Lower 0x07
3. Add R7 R4 R7

Note that all three of these instructions read and write to R7.

In any program, it was common to have two load-immediate instructions followed by an instruction that would read from R7 (the registers that load-immediate instruction uses). When the first load-immediate instruction was at the execute stage, the stall flag went high (to avoid RAW hazards as the next instruction being *LOADIMM.lower* needed to read from R7), and we simply just stopped the IDEX latch but did not clear the contents of the ALU and other units in the execute stage. This led to the instruction continuing to move through the pipeline and, additionally, becoming duplicated for as long as the stall bit was high. This was ok if only two instructions were reading from R7, but if there was a third instruction, there was enough time for the first instruction to propagate through and write back to the register file when the third instruction was decoded. The write-back of the duplicate first instruction would end up clearing the write-back pending flag that the second instruction set. This flag was set by the second instruction (loadimm.Lower 0x07) specifically so that the third instruction would wait to read from R7 until R7 was written to by the second instruction. So now, the R7-pending write-back flag raised by the second instruction is wrongfully lowered by the first instruction, which was duplicated during the stall, and the third instruction is wrongfully allowed to read from R7. In this way, a RAW hazard occurs. To fix this issue, we added extra functionality to the stall flag, in which if the stall flag is raised, the IDEX latch produces a NOPs after one clock cycle to avoid duplicating instruction. Additionally, we made the IDEX latch respond to the rising edge of the clock.

After fixing this problem, we found another issue, this time with the branch relative instructions. Specifically, as noted before, we implemented the branching control in the execute stage of the pipeline. Before implementing stalling, we found that to find the program counter to branch relative to (the program counter that came with the instruction, we simply had to subtract 4 from the current program counter. But once we stalled, the current program counter was now offset even more from the instruction at the execute stage, and as such, we would begin to branch relative from the wrong location. For this reason, we had to now make another program that would move through the pipeline along with the instruction being executed. In this way, the branch control logic would now always see the correct program counter, and we would branch from the correct location.

Finally, after integrating both the RAM and the ROM units into our design, we had to add special logic which would, based on the program counter, select whether the next instruction was to come from the RAM, or the ROM. This was no problem and we implemented this by making a 1:2 decoder circuit, whose outputs we controlled by the 11th bit of the program counter. If the program counter was above 1023, its 11th bit would go high, and this decoder would select the RAM unit to be read from, rather than the ROM unit. The problem we found in the end was that we forgot to implement a correction for the LOAD and STORE instruction. For example, if the loaded program were trying to store at address 1030 (address 6 in the RAM), we would not correct and find address 6 in the RAM, rather we'd look for address in RAM, which is not correct. In the end, we found this error too late (mere minutes before the presentation), we unfortunately could not fix this issue in time.

In the end, we presented our project to the TA, and our processor was evaluated. Physically, our processor was shown to be able to run every instruction in the boot loader code, even reading and writing through the in and out ports. However, once it had finished executing the boot loader code, it unfortunately had trouble running the new program stored in RAM. This was due to the issue described in the paragraph above.

The processor's overall speed was not measured due to lack of time, as we had not gotten a full solution by the time of the presentation. To measure the speed in simulation, we tried to get Vivado to do a timing analysis of the processor, but the timing information, specifically tco_min, tco_max, and other critical delay information, was not provided by the course, nor was it provided on our FPGA's (the XC7A35T-1CPG236C) datasheet. For this reason, we could not find the min slack time nor the max clock rate of our processor.

In terms of hardware utilization, the bar chart in Figure 8 in the results section shows the total hardware usage of the HPU system, including the RAM and the console, and then in purple, the RAM itself. The chart shows that the HPU used just a fraction of the available space on the FPGA we used, being the XC7A35T-1CPG236C, including the display module provided to us by the lab technician, Brent Sirna.

Additionally, this figure shows that the RAM took up the vast majority of the space within the system. If we were to design this for a smaller system, the RAM module would have the largest priority to shrink.

Conclusion

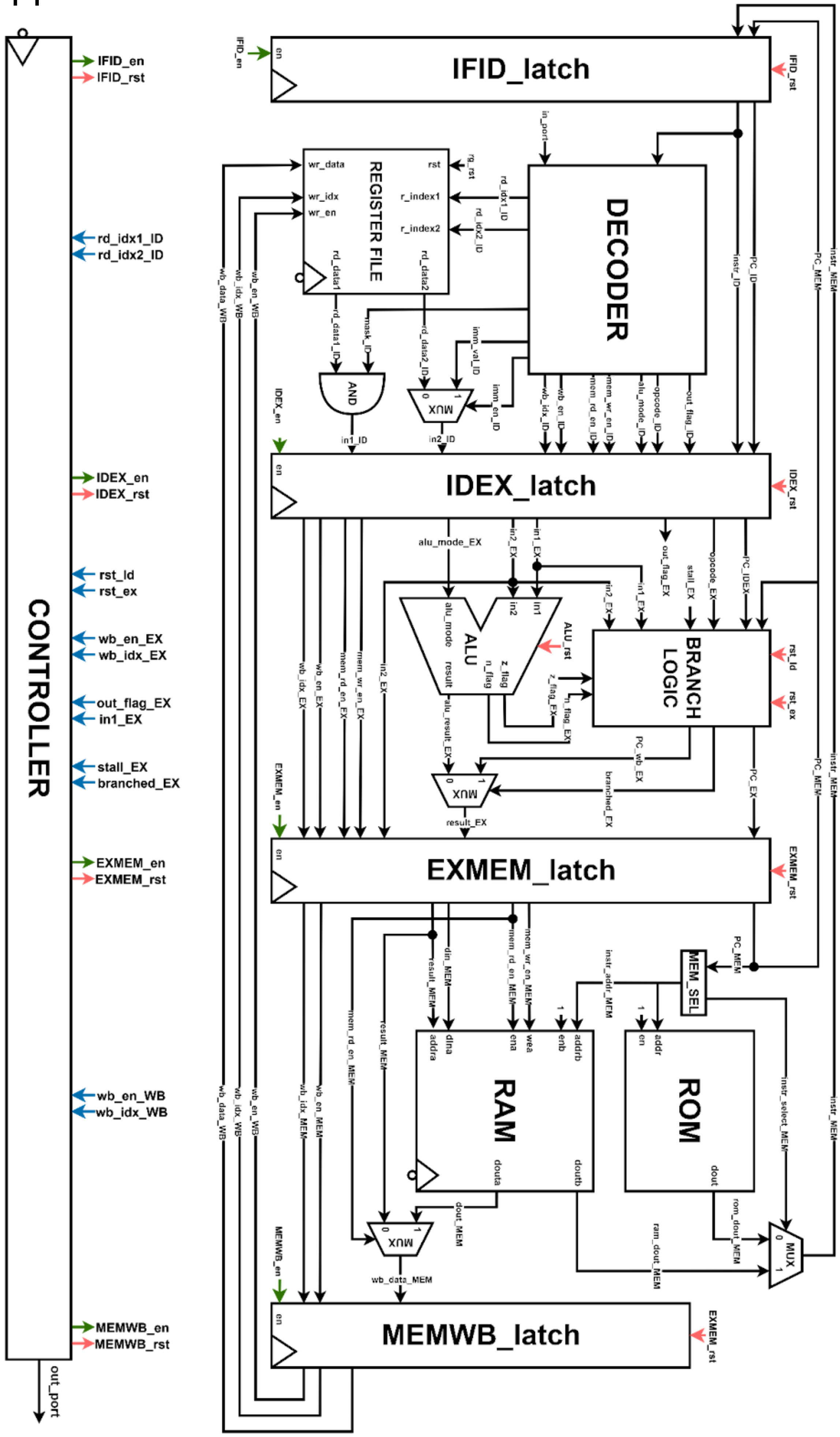
Over the duration of this semester, we used Vivado 2017.4 to build a 5-stage pipelined, 16-bit CPU, dubbed the Hazard516, or HPU (standing for Hazard Processing Unit). It was shown through simulations and hardware implementation that it could run all A, B, and L format instructions found in the ISA reference manual [1]. It was shown to complete all of the instructions given in the boot loader code but had trouble using a program stored in RAM. We were able to implement automatic RAW hazard control and pipeline flushing when branched. The RAW hazard control allowed for the programmer to not have to specify NOPs when writing assembly, increasing program development time and simplicity. The HPU could have benefited from the addition of push and pop handling and branch prediction, which would have increased the overall CPI of the processor.

In the end, we dove into the intricacies of CPU design and became very familiar with the workings of a basic pipeline. We learned about all the hazards that come with a pipeline and how to mitigate them. Additionally, we learned about the boot-loading process and how a CPU loads programs. And, of course, we became quite familiar with the Vivado EDA tool and the workings of the VHDL language. These skills will set us up well for a future job in CPU architecture or any position in digital design.

References

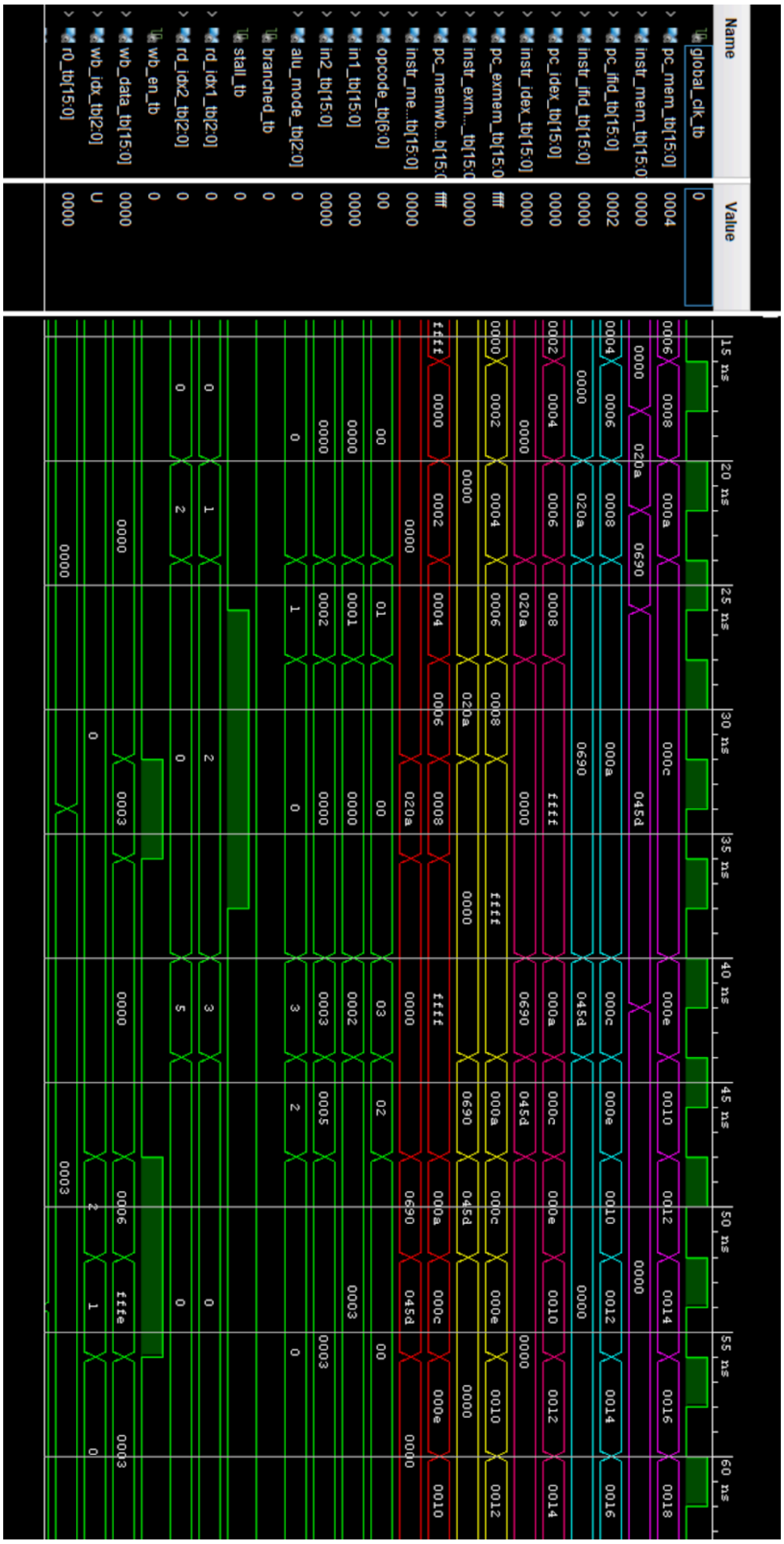
[1] “ECE 449: Instruction set (16-bit)”, University of Victoria Department of Electrical and Computer Engineering, https://www.engr.uvic.ca/~ece449/lab/index.html

Appendix A

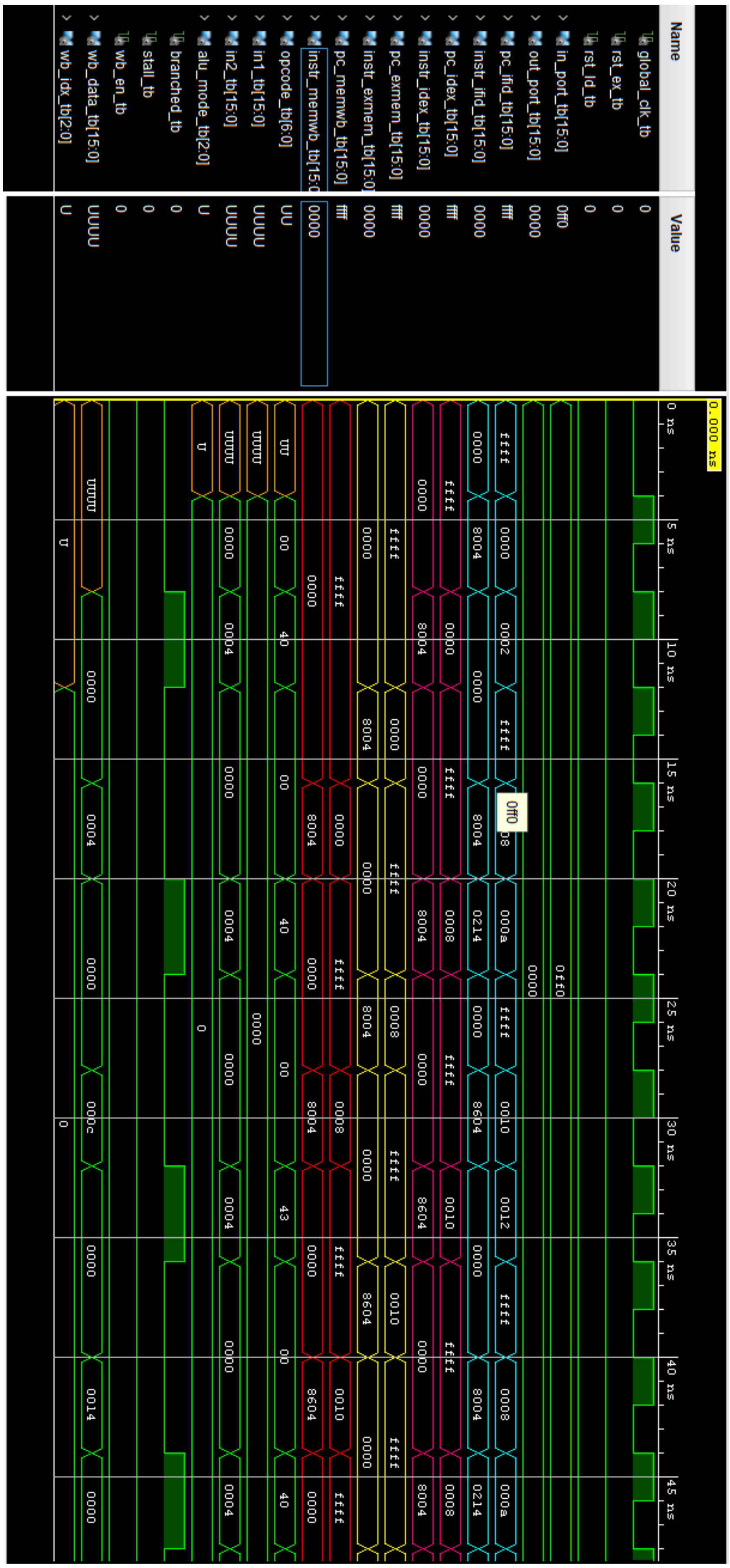


CONTROLLER
IFID_en
IFID_rst
rd_idx1_ID
rd_idx2_ID
IDEX_en
IDEX_rst
rst_id
rst_ex
wb_en_EX
wb_idx_EX
out_flag_EX
in1_EX
stall_EX
branched_EX
EXMEM_en
EXMEM_rst
wb_en_WB
wb_idx_WB
MEMWB_en
MEMWB_rst
out_port
IFID_latch
REGISTER FILE
DECODER
AND
MUX
IDEX_latch
ALU
BRANCH LOGIC
EXMEM_latch
RAM
ROM
MEM_SEL
MEMWB_latch

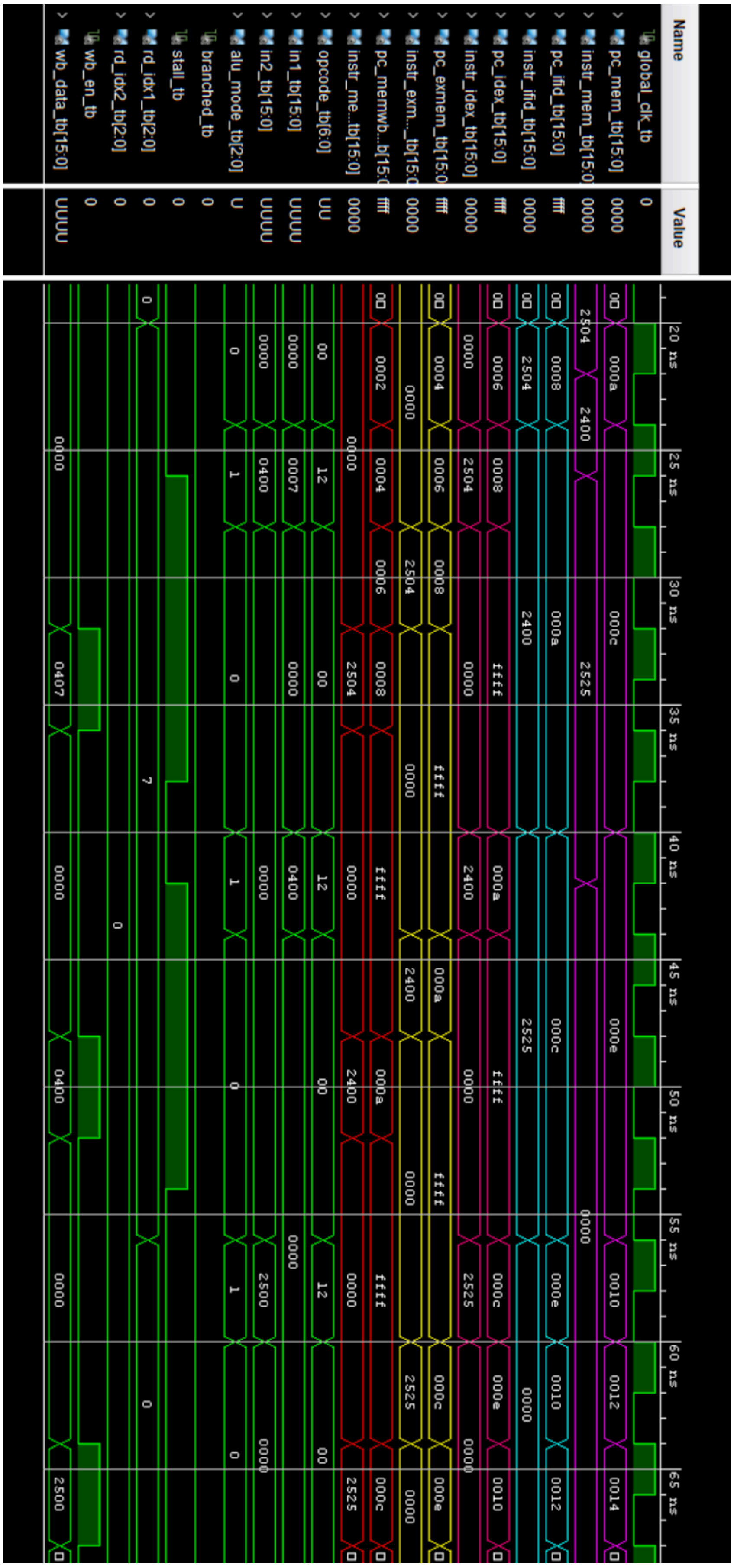
Appendix B



Format A timing diagram



Format B timing diagram



Format L Timing Diagram

Appendix C

Hpu.vhd

```
-------------------------------------------------------------------------
-----------
-- Company:
-- Engineer:
--
-- Create Date: 02/28/2025 10:14:01 AM
-- Design Name:
-- Module Name: hpu - Behavioral
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
-------------------------------------------------------------------------
-----------


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;


-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;
```

```
entity hpu is
   port (
    -- ## REAL IO
        global_clk : in STD_LOGIC;

        rst_ex : in STD_LOGIC;
        rst_ld : in STD_LOGIC;

        in_port : in STD_LOGIC_VECTOR(15 downto 0);

        out_port : out STD_LOGIC_VECTOR(15 downto 0);

        debug_console : in STD_LOGIC;
        board_clock: in std_logic;

        vga_red : out std_logic_vector( 3 downto 0 );
        vga_green : out std_logic_vector( 3 downto 0 );
        vga_blue : out std_logic_vector( 3 downto 0 );

        h_sync_signal : out std_logic;
        v_sync_signal : out std_logic


--      -- ## TESTBENCH IO

--          pc_ex_out : out STD_LOGIC_VECTOR(15 downto 0);
--          pc_mem_out : out STD_LOGIC_VECTOR(15 downto 0);

--          instr_MEM_out : out STD_LOGIC_VECTOR(15 downto 0);
--          instr_EX_out : out STD_LOGIC_VECTOR(15 downto 0);


--          pc_IFID_out : out STD_LOGIC_VECTOR(15 downto 0);
--          pc_IDEX_out : out STD_LOGIC_VECTOR(15 downto 0);
--          pc_EXMEM_out : out STD_LOGIC_VECTOR(15 downto 0);
--          pc_MEMWB_out : out STD_LOGIC_VECTOR(15 downto 0);

--          instr_IFID_out : out STD_LOGIC_VECTOR(15 downto 0);
--          instr_IDEX_out : out STD_LOGIC_VECTOR(15 downto 0);
--          instr_EXMEM_out : out STD_LOGIC_VECTOR(15 downto 0);
--          instr_MEMWB_out : out STD_LOGIC_VECTOR(15 downto 0);

--          n_flag     : out  std_logic;
--         z_flag     : out  std_logic;

--         opcode : out STD_LOGIC_VECTOR(6 downto 0);

--         in1 : out STD_LOGIC_VECTOR(15 downto 0);
--         in2 : out STD_LOGIC_VECTOR(15 downto 0);

--         alu_mode : out STD_LOGIC_VECTOR(2 downto 0);

--         rd_idx1,rd_idx2 : out STD_LOGIC_VECTOR(2 downto 0);

--         branched,stall : out STD_LOGIC;

--         wb_en : out STD_LOGIC;
--         wb_data : out STD_LOGIC_VECTOR(15 downto 0);
--         wb_idx : out STD_LOGIC_VECTOR(2 downto 0);

--         r0,r1,r2,r3,r4,r5,r6,r7: out STD_LOGIC_VECTOR(15 downto 0);

--         pending_wb_debug : out STD_LOGIC_VECTOR(7 downto 0)

    );

end hpu;

architecture brent of hpu is


-- ## INSTRUCTION DECODER and IDIF_LATCHES INTERMEDIARY SIGNALS ##
    signal instr_ID     : STD_LOGIC_VECTOR(15 downto 0);
    signal opcode_ID    : STD_LOGIC_VECTOR(6 downto 0);
    signal alu_mode_ID  : STD_LOGIC_VECTOR(2 downto 0);
    signal out_flag_ID : STD_LOGIC := '0';
    signal PC_ID      : STD_LOGIC_VECTOR(15 downto 0);


    signal wb_en_ID     : STD_LOGIC := '0';
    signal wb_idx_ID    : STD_LOGIC_VECTOR(2 downto 0);

    signal rd_idx1_ID   : STD_LOGIC_VECTOR(2 downto 0);
    signal rd_idx2_ID   : STD_LOGIC_VECTOR(2 downto 0);

    signal imm_val_ID   : STD_LOGIC_VECTOR(15 downto 0);
    signal imm_en_ID    : STD_LOGIC := '0';

    signal mem_wr_en_ID : STD_LOGIC := '0';
    signal mem_rd_en_ID : STD_LOGIC := '0';

    signal in1_ID,in2_ID : STD_LOGIC_VECTOR(15 downto 0);
    signal rd_data1_ID, rd_data2_ID : STD_LOGIC_VECTOR(15 downto 0);
    signal mask_ID : STD_LOGIC_VECTOR(15 downto 0)  := x"FFFF";


-- ## ALU, BRANCH and EXMEM_LATCHES INTERMEDIARY SIGNALS ##
    signal rst_ex_EX,rst_ld_EX : STD_LOGIC := '0';

    signal instr_IDEX : STD_LOGIC_VECTOR(15 downto 0);
    signal PC_IDEX : STD_LOGIC_VECTOR(15 downto 0);
    signal PC_EX : STD_LOGIC_VECTOR(15 downto 0);
    signal opcode_EX : STD_LOGIC_VECTOR(6 downto 0);
    signal alu_mode_EX : STD_LOGIC_VECTOR(2 downto 0);
    signal out_flag_EX : STD_LOGIC := '0';

    signal branched_EX,stall_EX : STD_LOGIC := '0';
    signal PC_wb_EX : STD_LOGIC_VECTOR(15 downto 0);


    signal mem_wr_en_EX : STD_LOGIC := '0';
    signal mem_rd_en_EX : STD_LOGIC := '0';

    signal wb_en_EX : STD_LOGIC := '0';
    signal wb_idx_EX : STD_LOGIC_VECTOR(2 downto 0);

    signal in1_EX, in2_EX : STD_LOGIC_VECTOR(15 downto 0);

    signal alu_result_EX : STD_LOGIC_VECTOR(15 downto 0);
    signal result_EX : STD_LOGIC_VECTOR(15 downto 0);

    signal z_flag_EX, n_flag_EX: STD_LOGIC := '0';



-- ## MEMORY and MEMWB_LATCHES INTERMEDIARY SIGNALS ##
    signal PC_MEM : STD_LOGIC_VECTOR(15 downto 0);

    signal PC_EXMEM : STD_LOGIC_VECTOR(15 downto 0);
    signal instr_EXMEM : STD_LOGIC_VECTOR(15 downto 0);

    signal PC_MEMWB : STD_LOGIC_VECTOR(15 downto 0);
    signal instr_MEMWB : STD_LOGIC_VECTOR(15 downto 0);

    signal instr_select_MEM : STD_LOGIC := '0';
    signal instr_addr_MEM : STD_LOGIC_VECTOR(15 downto 0);
    signal instr_MEM : STD_LOGIC_VECTOR(15 downto 0);

    signal mem_wr_en_MEM : STD_LOGIC := '0';
    signal mem_rd_en_MEM : STD_LOGIC := '0';
    signal mem_wr_en_MEM_0x0vec : STD_LOGIC_VECTOR(0 downto 0);

    signal din_MEM,dout_MEM : STD_LOGIC_VECTOR(15 downto 0);
    signal result_MEM : STD_LOGIC_VECTOR(15 downto 0);

    signal rom_dout_MEM : STD_LOGIC_VECTOR(15 downto 0);
    signal ram_dout_MEM : STD_LOGIC_VECTOR(15 downto 0);


    signal wb_en_MEM : STD_LOGIC := '0';
    signal wb_idx_MEM : STD_LOGIC_VECTOR(2 downto 0);
    signal wb_data_MEM : STD_LOGIC_VECTOR(15 downto 0);

    signal wb_en_WB : STD_LOGIC := '0';
    signal wb_idx_WB : STD_LOGIC_VECTOR(2 downto 0);
    signal wb_data_WB : STD_LOGIC_VECTOR(15 downto 0);





-- ## LATCHES & RST##
    signal IFID_en_FC: STD_LOGIC := '1';
    signal IFID_rst_FC : STD_LOGIC := '0';

    signal IDEX_en_FC: STD_LOGIC := '1';
    signal IDEX_rst_FC : STD_LOGIC := '0';

    signal EXMEM_en_FC: STD_LOGIC := '1';
    signal EXMEM_rst_FC: STD_LOGIC := '0';

    signal MEMWB_en_FC: STD_LOGIC := '1';
    signal MEMWB_rst_FC: STD_LOGIC := '0';

    signal rg_rst_FC : STD_LOGIC := '0';
    signal alu_rst_FC : STD_LOGIC := '0';

    signal out_port_FC : STD_LOGIC_VECTOR(15 downto 0);

    signal r0_i,r1_i,r2_i,r3_i,r4_i,r5_i,r6_i,r7_i :
STD_LOGIC_VECTOR(15 downto 0);
    signal pending_wb : STD_LOGIC_VECTOR(7 downto 0);


component console is
    port (

--
-- Stage 1 Fetch
--
        s1_pc : in STD_LOGIC_VECTOR ( 15 downto 0 );
        s1_inst : in STD_LOGIC_VECTOR ( 15 downto 0 );



--
-- Stage 2 Decode
--
        s2_pc : in STD_LOGIC_VECTOR ( 15 downto 0 );
        s2_inst : in STD_LOGIC_VECTOR ( 15 downto 0 );

        s2_reg_a : in STD_LOGIC_VECTOR( 2 downto 0 );
        s2_reg_b : in STD_LOGIC_VECTOR( 2 downto 0 );
        s2_reg_c : in STD_LOGIC_VECTOR( 2 downto 0 );

        s2_reg_a_data : in STD_LOGIC_VECTOR( 15 downto 0 );
        s2_reg_b_data : in STD_LOGIC_VECTOR( 15 downto 0 );
        s2_reg_c_data : in STD_LOGIC_VECTOR( 15 downto 0 );

        s2_immediate : in STD_LOGIC_VECTOR( 15 downto 0 );



--
-- Stage 3 Execute
--
        s3_pc : in STD_LOGIC_VECTOR ( 15 downto 0 );
        s3_inst : in STD_LOGIC_VECTOR ( 15 downto 0 );

        s3_reg_a : in STD_LOGIC_VECTOR( 2 downto 0 );
        s3_reg_b : in STD_LOGIC_VECTOR( 2 downto 0 );
        s3_reg_c : in STD_LOGIC_VECTOR( 2 downto 0 );

        s3_reg_a_data : in STD_LOGIC_VECTOR( 15 downto 0 );
        s3_reg_b_data : in STD_LOGIC_VECTOR( 15 downto 0 );
        s3_reg_c_data : in STD_LOGIC_VECTOR( 15 downto 0 );

        s3_immediate : in STD_LOGIC_VECTOR( 15 downto 0 );

--
-- Branch and memory operation
--
        s3_r_wb : in STD_LOGIC;
        s3_r_wb_data : in STD_LOGIC_VECTOR( 15 downto 0 );

        s3_br_wb : in STD_LOGIC;
        s3_br_wb_address : in STD_LOGIC_VECTOR( 15 downto 0 );

        s3_mr_wr : in STD_LOGIC;
        s3_mr_wr_address : in STD_LOGIC_VECTOR( 15 downto 0 );
        s3_mr_wr_data : in STD_LOGIC_VECTOR( 15 downto 0 );

        s3_mr_rd : in STD_LOGIC;
        s3_mr_rd_address : in STD_LOGIC_VECTOR( 15 downto 0 );

--
-- Stage 4 Memory
--
        s4_pc : in STD_LOGIC_VECTOR( 15 downto 0 );
        s4_inst : in STD_LOGIC_VECTOR( 15 downto 0 );

        s4_reg_a : in STD_LOGIC_VECTOR( 2 downto 0 );

        s4_r_wb : in STD_LOGIC;
        s4_r_wb_data : in STD_LOGIC_VECTOR( 15 downto 0 );

--
-- CPU registers
--
```

```
        register_0 : in STD_LOGIC_VECTOR ( 15 downto 0 );
        register_1 : in STD_LOGIC_VECTOR ( 15 downto 0 );
        register_2 : in STD_LOGIC_VECTOR ( 15 downto 0 );
        register_3 : in STD_LOGIC_VECTOR ( 15 downto 0 );
        register_4 : in STD_LOGIC_VECTOR ( 15 downto 0 );
        register_5 : in STD_LOGIC_VECTOR ( 15 downto 0 );
        register_6 : in STD_LOGIC_VECTOR ( 15 downto 0 );
        register_7 : in STD_LOGIC_VECTOR ( 15 downto 0 );

--
-- CPU registers overflow flags
--
        register_0_of : in STD_LOGIC;
        register_1_of : in STD_LOGIC;
        register_2_of : in STD_LOGIC;
        register_3_of : in STD_LOGIC;
        register_4_of : in STD_LOGIC;
        register_5_of : in STD_LOGIC;
        register_6_of : in STD_LOGIC;
        register_7_of : in STD_LOGIC;

--
-- CPU Flags
--
        zero_flag : in STD_LOGIC;
        negative_flag : in STD_LOGIC;
        overflow_flag : in STD_LOGIC;

--
-- Debug screen enable
--
        debug : in STD_LOGIC;

--
-- Text console display memory access signals ( clk is the processor
clock )
--
        addr_write : in  STD_LOGIC_VECTOR (15 downto 0);
        clk : in  STD_LOGIC;
        data_in : in  STD_LOGIC_VECTOR (15 downto 0);
        en_write : in  STD_LOGIC;

--
```

```
-- Video related signals
--
        board_clock : in STD_LOGIC;
        v_sync_signal : out STD_LOGIC;
        h_sync_signal : out STD_LOGIC;
        vga_red : out STD_LOGIC_VECTOR( 3 downto 0 );
        vga_green : out STD_LOGIC_VECTOR( 3 downto 0 );
        vga_blue : out STD_LOGIC_VECTOR( 3 downto 0 )

    );
end component;




begin
    mem_wr_en_MEM_0x0vec  <= (0=>mem_wr_en_MEM);

    IFID_latch : entity work.IFID_latches
        port map (
            clk => global_clk, en => IFID_en_FC, rst => IFID_rst_FC,
            PC_in=>PC_MEM,
            PC_out=>PC_ID,
            instr_in=>instr_MEM,
            instr_out=>instr_ID
        );


    decoder : entity work.instr_decoder
        port map(
            instr      => instr_ID,
            stall      => stall_EX,

            opcode     => opcode_ID,
            alu_mode   => alu_mode_ID,

            out_flag   => out_flag_ID,

            wb_en      => wb_en_ID,
            wb_idx     => wb_idx_ID,

            rd_idx1    => rd_idx1_ID,
            rd_idx2    => rd_idx2_ID,
            imm_val    => imm_val_ID,
            imm_en     => imm_en_ID,

            mem_wr_en  => mem_wr_en_ID,
            mem_rd_en  => mem_rd_en_ID,

            mask => mask_ID,
            in_port => in_port
        );


    registerfile : entity work.register_file
        port map (
            rst=> rg_rst_FC, clk=>global_clk,

            rd_index1=> rd_idx1_ID, rd_index2 => rd_idx2_ID,
            rd_data1=> rd_data1_ID, rd_data2 => rd_data2_ID,

            wr_index => wb_idx_WB,
            wr_data => wb_data_WB,
            wr_en => wb_en_WB,
            r0 => r0_i, r1 => r1_i, r2 => r2_i, r3 => r3_i, r4 => r4_i,
r5 => r5_i, r6 => r6_i, r7 => r7_i
        );


    imm_MUX : entity work.mux
        port map(
            en => imm_en_ID,
            in0 => rd_data2_ID,
            in1 => imm_val_ID,
            output => in2_ID
        );

    and16bit : entity work.and_gate
        port map(
            in1 => mask_ID,
            in2 => rd_data1_ID,
            output => in1_ID
        );

    IDEX_latch : entity work.IDEX_LATCHES
```

```
        port map(
            clk=>global_clk, en=> IDEX_en_FC, rst => IDEX_rst_FC,

            pc_in => PC_ID, pc_out => pc_IDEX,
            instr_in => instr_ID, instr_out => instr_IDEX,

            opcode_in => opcode_ID, opcode_out => opcode_EX,
            out_flag_in => out_flag_ID, out_flag_out => out_flag_EX,

            mem_wr_en_in => mem_wr_en_ID, mem_rd_en_in => mem_rd_en_ID,
            mem_wr_en_out => mem_wr_en_EX, mem_rd_en_out =>
mem_rd_en_EX,

            wb_en_in =>wb_en_ID, wb_idx_in => wb_idx_ID,
            wb_en_out => wb_en_EX, wb_idx_out => wb_idx_EX,

            alu_mode_in =>  alu_mode_ID,
            alu_mode_out =>  alu_mode_EX,

            in1_in => in1_ID, in2_in => in2_ID,
            in1_out => in1_EX, in2_out => in2_EX
        );

    branch : entity work.branch_logic
        port map(
            rst_ex => rst_ex,rst_ld => rst_ld,

            stall=>stall_EX,

            PC_in => PC_MEM,
            PC_EX => PC_IDEX,
            opcode_in => opcode_EX,
            in1_in => in1_EX, in2_in => in2_EX,
            z_flag_in => z_flag_EX, n_flag_in => n_flag_EX,

            PC_out => PC_EX,
            wb_out => PC_wb_EX,
            branched_out => branched_EX
        );

    alu : entity work.alu
        port map(
            in1 => in1_EX, in2 => in2_EX,
            alu_mode => alu_mode_EX, rst=> alu_rst_FC,

            result => alu_result_EX,
            z_flag => z_flag_EX, n_flag => n_flag_EX
        );

    result_MUX : entity work.mux
        port map(
            en => branched_EX,
            in0 => alu_result_EX,
            in1 => PC_wb_EX,
            output => result_EX
        );


    EXMEM_latch : entity work.EXMEM_latches
        port map(
            clk => global_clk, en => EXMEM_en_FC, rst => EXMEM_rst_FC,

            PC_in => PC_EX, PC_out => PC_MEM,

            PC_tb_in => PC_IDEX, PC_tb_out => PC_EXMEM,
            instr_in => instr_IDEX, instr_out => instr_EXMEM,

            mem_wr_en_in => mem_wr_en_EX, mem_rd_en_in => mem_rd_en_EX,
            mem_wr_en_out => mem_wr_en_MEM, mem_rd_en_out =>
mem_rd_en_MEM,

            wb_en_in =>wb_en_EX, wb_idx_in => wb_idx_EX,
            wb_en_out => wb_en_MEM, wb_idx_out => wb_idx_MEM,

            din_in => in2_EX, din_out => din_MEM,
            result_in => result_EX, result_out => result_MEM
        );


    mem_sel : entity work.mem_sel
        port map(
            pc => PC_MEM,
            addr => instr_addr_MEM,
            mem_select => instr_select_MEM
        );
```

```
ram : entity work.dummy_ram
    port map(
        clk => global_clk,

        addra => result_MEM,
        ena => mem_rd_en_MEM,
        douta => dout_MEM,

        addrb => instr_addr_MEM,
        enb => '1',
        doutb => ram_dout_MEM,

        wea => mem_wr_en_MEM,
        dina => din_MEM
    );


rom : entity work.dummy_rom
    port map(
        clk => global_clk,

        addr => instr_addr_MEM,
        en => '1',
        dout => rom_dout_MEM
    );


instr_MUX : entity work.mux
    port map(
        en => instr_select_MEM,
        in0 => rom_dout_MEM,
        in1 => ram_dout_MEM,
        output => instr_MEM
    );


wb_MUX : entity work.mux
    port map(
        en => mem_rd_en_MEM,
        in0 => result_MEM,
        in1 => dout_MEM,
        output => wb_data_MEM
        );


    MEMWB_latch : entity work.MEMWB_latches
        port map(
            clk => global_clk, en => MEMWB_en_FC, rst => MEMWB_rst_FC,

            PC_in => PC_EXMEM, PC_out => PC_MEMWB,
            instr_in => instr_EXMEM, instr_out => instr_MEMWB,

            wb_en_in =>wb_en_MEM, wb_en_out => wb_en_WB,
            wb_idx_in => wb_idx_MEM, wb_idx_out => wb_idx_WB,

            wb_data_in => wb_data_MEM, wb_data_out => wb_data_WB
        );


   controller : entity work.controller
        port map(
            rst_ex => rst_ex,rst_ld => rst_ld,

            clk         => global_clk,
            PC          => PC_MEM,

            stall       => stall_EX,

            wb_en_ID    => wb_en_ID,
            wb_en_EX    => wb_en_EX,
            wb_en_WB    => wb_en_WB,
            wb_idx_ID   => wb_idx_ID,
            wb_idx_EX   => wb_idx_EX,
            wb_idx_WB   => wb_idx_WB,
            rd_idx1_ID  => rd_idx1_ID,
            rd_idx2_ID  => rd_idx2_ID,

            in1         => in1_EX,
            out_flag    => out_flag_EX,

            branched    => branched_EX,

            out_port    => out_port_FC,

            IFID_en     => IFID_en_FC,
            IDEX_en     => IDEX_en_FC,
            EXMEM_en    => EXMEM_en_FC,
            MEMWB_en    => MEMWB_en_FC,

            rg_rst      => rg_rst_FC,
            alu_rst     => alu_rst_FC,

            IFID_rst    => IFID_rst_FC,
            IDEX_rst    => IDEX_rst_FC,
            EXMEM_rst   => EXMEM_rst_FC,
            MEMWB_rst   => MEMWB_rst_FC,

            pending_wb_debug => pending_wb
        );


    console_display : console
        port map(
        --
        -- Stage 1 Fetch
        --
            s1_pc     => PC_MEM,
            s1_inst   => instr_MEM,

        --
        -- Stage 2 Decode
        --

            s2_pc     => PC_ID,
            s2_inst   => instr_ID,

            s2_reg_a => wb_idx_ID,
            s2_reg_b => rd_idx1_ID,
            s2_reg_c => rd_idx2_ID,

            s2_reg_a_data => (others => '0'),
            s2_reg_b_data => rd_data1_ID,
            s2_reg_c_data => rd_data2_ID,
            s2_immediate    => imm_val_ID,

        --
        -- Stage 3 Execute
        --
```

```
            s3_pc     => PC_IDEX,
            s3_inst   => instr_IDEX,

            s3_reg_a => alu_mode_EX,
            s3_reg_b => (others => '0'),
            s3_reg_c => (others => '0'),

            s3_reg_a_data => (others => '0'),
            s3_reg_b_data => in1_EX,
            s3_reg_c_data => in2_EX,
            s3_immediate  => (others => '0'),

            s3_r_wb       => wb_en_EX,
            s3_r_wb_data  => result_EX,

            s3_br_wb         => branched_EX,
            s3_br_wb_address => PC_wb_EX,

            s3_mr_wr         => mem_wr_en_EX,
            s3_mr_wr_address => result_EX,
            s3_mr_wr_data    => in2_EX,

            s3_mr_rd         => mem_rd_en_EX,
            s3_mr_rd_address => result_EX,


        --
        -- Stage 4 Memory
        --

            s4_pc      => PC_EXMEM,
            s4_inst    => instr_EXMEM,
            s4_reg_a   => wb_idx_MEM,
            s4_r_wb    => wb_en_MEM,
            s4_r_wb_data => wb_data_MEM,

        --
        -- CPU registers
        --

            register_0 => r0_i,
            register_1 => r1_i,
            register_2 => r2_i,
            register_3 => r3_i,
            register_4 => r4_i,
            register_5 => r5_i,
            register_6 => r6_i,
            register_7 => r7_i,

            register_0_of => '0',
            register_1_of => '0',
            register_2_of => '0',
            register_3_of => '0',
            register_4_of => '0',
            register_5_of => '0',
            register_6_of => '0',
            register_7_of => '0',

        --
        -- CPU Flags
        --
            zero_flag     => z_flag_EX,
            negative_flag => n_flag_EX,
            overflow_flag => '0',


    --
    -- Debug screen enable
    --
            debug => debug_console,


    --
    -- Text console display memory access signals ( clk is the
processor clock )
    --


        clk => global_clk,
        addr_write => x"0000",
        data_in => x"0000",
        en_write => '0',


    --
    -- Video related signals
    --
```

```
        board_clock => board_clock,
        h_sync_signal => h_sync_signal,
        v_sync_signal => v_sync_signal,
        vga_red => vga_red,
        vga_green => vga_green,
        vga_blue => vga_blue
    );



out_port <= out_port_FC;





---- TESTBENCH IO
--instr_MEM_out <= instr_MEM;

--instr_IFID_out <= instr_ID;
--instr_IDEX_out <= instr_IDEX;
--instr_EXMEM_out <= instr_EXMEM;
--instr_MEMWB_out <= instr_MEMWB;

--pc_MEM_out <= PC_MEM;
--pc_EX_out <= PC_EX;
--pc_IFID_out <= PC_ID;
--pc_IDEX_out <= PC_IDEX;
--pc_EXMEM_out <= PC_EXMEM;
--pc_MEMWB_out <= PC_MEMWB;

--n_flag <= n_flag_EX;
--z_flag <= z_flag_EX;

--opcode <= opcode_EX;

--rd_idx1 <= rd_idx1_ID;
--rd_idx2  <= rd_idx2_ID;

--stall <= stall_EX;

---- ALU Inputs
--in1 <= in1_EX;
--in2  <= in2_EX;
```

```
--alu_mode <= alu_mode_EX;

--branched <= branched_EX;

---- Write-Back Stage Outputs
--wb_en <= wb_en_WB;
--wb_data <= wb_data_WB;
--wb_idx <= wb_idx_WB;

--r0 <= r0_i;
--r1 <= r1_i;
--r2 <= r2_i;
--r3 <= r3_i;
--r4 <= r4_i;
--r5 <= r5_i;
--r6 <= r6_i;
--r7 <= r7_i;

--pending_wb_debug <= pending_wb;

end architecture brent; -- NOOOOOOOO
```

alu.vhd

```
ibrary IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all; -- use that, it's a better coding guideline

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity alu is
port (
    in1,in2 : in STD_LOGIC_VECTOR(15 downto 0);
```

```vhdl
    alu_mode : in STD_LOGIC_VECTOR(2 downto 0);

    rst : in STD_LOGIC;

    result   : out STD_LOGIC_VECTOR(15 downto 0);
    z_flag   : out STD_LOGIC;
    n_flag   : out STD_LOGIC
    );
end alu;

architecture Behavioral of alu is
    signal z_flag_i,n_flag_i : std_logic := '0';

begin

    process(in1,in2,alu_mode,rst)

    variable product: std_logic_vector(31 downto 0) := (others => '0');
    variable temp: std_logic_vector(15 downto 0) := (others => '0');
    variable zeros: std_logic_vector(15 downto 0) := (others => '0');


    begin

    if (rst = '1') then
        z_flag_i <= '0';
        n_flag_i <= '0';
        result <= zeros;

    else
        if(alu_mode = "000") then
            -- NO OP
            temp := in1;

        elsif(alu_mode = "001") then
            -- ADD
            temp := STD_LOGIC_VECTOR(signed(in1) + signed(in2));

        elsif(alu_mode = "010") then
            -- SUB
            temp := STD_LOGIC_VECTOR(signed(in1) - signed(in2));

        elsif(alu_mode = "011") then
```

```
            --MULT
            product := STD_LOGIC_VECTOR(signed(in1) * signed(in2));
            temp := product (15 downto 0);

        elsif(alu_mode = "100") then
            -- NAND
            temp := STD_LOGIC_VECTOR(signed(in1) NAND signed(in2));

        elsif(alu_mode = "101") then
             --SHIFT LEFT
             temp :=
STD_LOGIC_VECTOR(shift_left(signed(in1),to_integer(signed(in2))));
--UPDATE TO SHIFT BY AN INPUT VALUE

        elsif(alu_mode = "110") then
            --SHIFT RIGHT
             temp :=
STD_LOGIC_VECTOR(shift_right(signed(in1),to_integer(signed(in2))));
--UPDATE TO SHIFT BY AN INPUT VALUE

        elsif(alu_mode = "111") then
            -- TEST
            temp := in1;

            if (temp = zeros) then
                z_flag_i <= '1';
            else
                z_flag_i <= '0';
            end if;

            if (signed(temp) < 0) then
                n_flag_i <= '1';
            else
                n_flag_i <= '0';
            end if;

        else
            temp := zeros;
        end if;

        result <= temp;
```

```
    end if;

n_flag <= n_flag_i;
z_flag <= z_flag_i;
end process;


end Behavioral;
```

And.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity and_gate is
 Port (
   -- INPUTS
   in1 : in STD_LOGIC_VECTOR(15 downto 0);
   in2 : in STD_LOGIC_VECTOR(15 downto 0);

  -- OUTPUTS
   output : out STD_LOGIC_VECTOR(15 downto 0)
   );
end and_gate;

architecture Behavioral of and_gate is

begin

    output <= in1 and in2;


```

```vhdl
end Behavioral;
```

Branch_logic.vhd

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity branch_logic is
    Port (
        rst_ex,rst_ld,stall : in STD_LOGIC;

        PC_in,PC_EX,in1_in,in2_in : in STD_LOGIC_VECTOR(15 downto 0);
        opcode_in : in STD_LOGIC_VECTOR(6 downto 0);

        n_flag_in,z_flag_in : in STD_LOGIC;

        PC_out,wb_out : out STD_LOGIC_VECTOR(15 downto 0);
        branched_out : out STD_LOGIC
    );
end branch_logic;

architecture Behavioral of branch_logic is

signal PC : std_logic_vector(15 downto 0) := (others => '0');
signal branched : std_logic := '0';
signal prev_stall : std_logic := '0';

begin
```

```
process(PC_in,PC_EX,in1_in,in2_in,opcode_in,n_flag_in,z_flag_in,rst_ld,
rst_ex,stall)
    begin

        if rst_ex = '1' then
            PC <= x"0000";
            branched <= '1';

        elsif rst_ld = '1' then
            PC <= x"0002";
            branched <= '1';


        elsif stall = '1' then

            PC <= PC_in;
            branched <= '0';

        else
            case to_integer(unsigned(opcode_in)) is

                when 64 =>
                    -- BRR

                        PC <= std_logic_vector(signed(PC_EX) +
shift_left(resize(signed(in2_in(14 downto 0)), 16), 1));
                        branched <= '1';

                when 65 =>
                    -- BRR.N
                    if n_flag_in = '1' then
                        PC <= std_logic_vector(signed(PC_EX) +
shift_left(resize(signed(in2_in(14 downto 0)), 16), 1));
                        branched <= '1';
                    else
                        PC <= STD_LOGIC_VECTOR(signed(PC_in) + 2);
                        branched <= '0';
                    end if;

                when 66 =>
                    -- BRR.Z
                    if z_flag_in = '1' then
                        PC <= std_logic_vector(signed(PC_EX) +
shift_left(resize(signed(in2_in(14 downto 0)), 16), 1));
                        branched <= '1';
                    else
                        PC <= STD_LOGIC_VECTOR(signed(PC_in) + 2);
                        branched <= '0';
                    end if;

                when 67 =>
                    -- BR
                        PC <= STD_LOGIC_VECTOR(signed(in1_in) +
shift_left(resize(signed(in2_in(14 downto 0)), 16), 1));
                    branched <= '1';

                when 68 =>
                    -- BR.N
                    if n_flag_in = '1' then
                        PC <= STD_LOGIC_VECTOR(signed(in1_in) +
shift_left(resize(signed(in2_in(14 downto 0)), 16), 1));
                        branched <= '1';
                    else
                        PC <= STD_LOGIC_VECTOR(signed(PC_in) + 2);
                        branched <= '0';
                    end if;
                when 69 =>
                    -- BR.Z
                    if z_flag_in = '1' then
                        PC <= STD_LOGIC_VECTOR(signed(in1_in) +
shift_left(resize(signed(in2_in(14 downto 0)), 16), 1));
                        branched <= '1';
                    else
                        PC <= STD_LOGIC_VECTOR(signed(PC_in) + 2);
                        branched <= '0';
                    end if;

                when 70 =>
                    -- BR.SUB (WB needs to be enabled from controller)
                    PC <= STD_LOGIC_VECTOR(signed(in1_in) +
shift_left(resize(signed(in2_in(14 downto 0)), 16), 1));
                    branched <= '1';

                when 71 =>
```

```
                    -- RETURN
                    PC <= STD_LOGIC_VECTOR(signed(in1_in));
                    branched <= '1';

                when others =>
                    -- NO BRANCH OP
                    PC <= STD_LOGIC_VECTOR(signed(PC_in) + 2);
                    branched <= '0';
                end case;
            end if;
    end process;

    PC_out <= PC;
    branched_out <= branched;
    wb_out <= STD_LOGIC_VECTOR(signed(PC_EX) + 2);

end Behavioral;
```

Controller.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity controller is
    port (
    -- ## INPUTS
        -- ## GLOBAL
            clk,rst_ld,rst_ex : in std_logic;
            PC : in std_logic_vector(15 downto 0);
        -- ## REGISTER MONITORING
            wb_en_ID,wb_en_EX,wb_en_WB : in std_logic;
            wb_idx_ID,wb_idx_EX,wb_idx_WB : in std_logic_vector(2
downto 0);
            rd_idx1_ID,rd_idx2_ID : in std_logic_vector(2 downto 0);

        -- ## OUT_PORT MONITORING
            in1 : in std_logic_vector(15 downto 0);
            out_flag : in std_logic;

        -- ## BRANCH MONITORING
            branched : in std_logic;


    -- ## OUTPUTS
        -- ## CPU OUT_PORT
            out_port : out std_logic_vector(15 downto 0);

        -- ## ENABLES
            IFID_en,IDEX_en,EXMEM_en,MEMWB_en,stall : out std_logic;

        -- ## RESETS
            rg_rst,alu_rst : out std_logic;
            IFID_rst,IDEX_rst,EXMEM_rst,MEMWB_rst : out std_logic;


    -- ## DEBUGGING
        pending_wb_debug : out std_logic_vector(7 downto 0)
     );


end controller;

architecture Behavioral of controller is

signal out_port_i : std_logic_vector(15 downto 0) := (others => '0');

signal IFID_en_i : std_logic := '1';
signal IDEX_en_i : std_logic := '1';
signal EXMEM_en_i : std_logic := '1';
signal MEMWB_en_i : std_logic := '1';

signal IFID_rst_i : std_logic := '0';
signal IDEX_rst_i : std_logic := '0';
signal EXMEM_rst_i : std_logic := '0';
signal MEMWB_rst_i : std_logic := '0';
signal rg_rst_i : std_logic := '0';
signal alu_rst_i : std_logic := '0';

signal stall_i : std_logic := '0';

signal pending_wb : std_logic_vector(7 downto 0) := (others => '0');


begin
    process(clk) begin

        if out_flag = '1' then
            out_port_i <= in1;
        end if;


        if not rising_edge(clk) then

            stall_i <= '0';
            IFID_en_i <= '1';
            IDEX_en_i <= '1';

            IFID_rst_i <= '0';
            IDEX_rst_i <= '0';


            -- CLEAR BUSY FLAG ON INCOMING WB
            if wb_en_WB = '1' then
               pending_wb(to_integer(unsigned(wb_idx_WB))) <= '0';
            end if;

            -- SET PENDING IF INSTR MADE IT TO EX
            if wb_en_EX = '1' then
               pending_wb(to_integer(unsigned(wb_idx_EX))) <= '1';


               if wb_idx_EX = rd_idx1_ID or wb_idx_EX = rd_idx2_ID then
                   stall_i <= '1';
                   IFID_en_i <= '0';
                   IDEX_rst_i <= '1';
               end if;

            end if;


            if branched = '1' then
                IFID_rst_i <= '1';
                IDEX_rst_i <= '1';

            elsif pending_wb(to_integer(unsigned(rd_idx1_ID))) = '1' or
            pending_wb(to_integer(unsigned(rd_idx2_ID))) = '1' then

                stall_i <= '1';
                IFID_en_i <= '0';
                IDEX_rst_i <= '1';

            end if;
        end if;
    end process;

    out_port  <= out_port_i;

    stall <= stall_i;

    IFID_en   <= IFID_en_i;
    IDEX_en   <= IDEX_en_i;
    EXMEM_en  <= EXMEM_en_i;
    MEMWB_en  <= MEMWB_en_i;

    IFID_rst  <= IFID_rst_i;
    IDEX_rst  <= IDEX_rst_i;
    EXMEM_rst <= EXMEM_rst_i;
    MEMWB_rst <= MEMWB_rst_i;

    rg_rst    <= rg_rst_i;
    alu_rst   <= alu_rst_i;

    pending_wb_debug <= pending_wb;
end Behavioral;
```

Dummy_ram.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all; -- use that, it's a better coding guideline

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity dummy_ram is
    port (
        clk    : in std_logic;

        addra,addrb   : in std_logic_vector(15 downto 0);

        ena,enb,wea     : in std_logic;
        dina    : in std_logic_vector(15 downto 0);

        douta   : out std_logic_vector(15 downto 0);
        doutb   : out std_logic_vector(15 downto 0)
    );
end entity dummy_ram;

architecture dummy_ram of dummy_ram is


-----------------------------------------------------------------------
-----
    -- Memory array for storing up to 256 words (16-bit each).
    -- Here we explicitly set the first 8 locations to some dummy
instructions.

-----------------------------------------------------------------------
-----
    type mem_array is array (0 to 1023) of std_logic_vector(15 downto
0);
```

```vhdl
    signal mem : mem_array := (others => (others => '0'));

begin


-----------------------------------------------------------------------
-----
    -- Single clocked process handling both Port A (read/write) and
Port B (read)

-----------------------------------------------------------------------
-----
    process(clk)
    begin
        if not rising_edge(clk) then

            if wea = '1' then
                mem(to_integer(unsigned(addra(7 downto 0)))) <= dina;
            end if;

            if ena = '1' then
                douta <= mem(to_integer(unsigned(addra(7 downto 0))));
            else
                douta <= (others => '0');
            end if;


            if enb = '1' then
                doutb <= mem(to_integer(unsigned(addrb(7 downto 0))));
            else
                doutb <= (others => '0');
            end if;

        end if;
    end process;

end architecture dummy_ram;
```

Dummy_rom.vhd

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
```

```
-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity dummy_rom is
    Port ( en : in STD_LOGIC;
           addr : in STD_LOGIC_VECTOR (15 downto 0);
           dout : out STD_LOGIC_VECTOR (15 downto 0);
           clk : in STD_LOGIC);
end dummy_rom;

architecture Behavioral of dummy_rom is

type mem_array is array (0 to 1023) of std_logic_vector(15 downto 0);
signal mem : mem_array := (

--     0000 => "1000000000000100", -- 1000000 000000100
brr           8
--     0008 => "1000000000000100", -- 1000000 000000100
brr           16
--     0010 => "0000001000010100", -- 1000000 000000100
brr           16
--     0016 => "1000011000000100", -- 1000011 000 000100
br            8

--     0008 => "0000001000001010", -- R0 <- 1 + 2
--     0010 => "0000011010010000", -- R2 < 2 * R0
--     0012 => "0000010001011101", -- R1 < R3 - R5
--     0014 => "0000111001000000", -- TEST R1

--     0088 => "0010010100000000", -- 0058 - 2500 ResetLoad:
loadimm.upper 0x00
--     0090 => "0010010010000000", -- 005A - 2480
loadimm.lower 0x80
--     0092 => "0010011110111000", -- 005C - 27B8
mov           r6,r7
```

```
--    0094 => "0100001010000000", -- 005E - 4280 WaitFor_AA:         in
r2
--    0096 => "0000100010010110", -- 0060 - 0896
nand          r2,r2,r6
--    0098 => "0000100010010010", -- 0062 - 0892
nand          r2,r2,r2
--    0100 => "0000111010000000", -- 0064 - 0E80
test          r2
--    0102 => "1000010111111100", -- 0066 - 85FC
brr.z         WaitFor_AA
--    0104 => "0010010111111111", -- 0068 - 25FF
loadimm.upper 0xFF
--    0106 => "0010010000000000", -- 006A - 2400
loadimm.lower 0x00



    0000 => "1000000000000100", -- 0000 - 8004                         brr
ResetExecute
    0002 => "1000000000101011", -- 0002 - 802B                         brr
ResetLoad
    0004 => "1000000000000001", -- 0004 - 8001                         brr
Interrupt
    0006 => "1000000000000000", -- 0006 - 8000 WaitForever:            brr
WaitForever
    0008 => "0010010100000100", -- 0008 - 2504 ResetExecute:
loadimm.upper BootVector.hi
    0010 => "0010010000000000", -- 000A - 2400
loadimm.lower BootVector.lo
    0012 => "0010000111111000", -- 000C - 21F8                         load
r7,r7
    0014 => "0000000000000000", -- 000E - 0000                         nop
    0016 => "0000000000000000", -- 0010 - 0000                         nop
    0018 => "0010010000000000", -- 0012 - 2400
loadimm.lower 0x00
    0020 => "0010011010111000", -- 0014 - 26B8                         mov
r2,r7
    0022 => "0010010100100101", -- 0016 - 2525
loadimm.upper 0x25
    0024 => "0010010000000000", -- 0018 - 2400
loadimm.lower 0x00
    0026 => "0000010010010111", -- 001A - 0497                         sub
r2,r2,r7
```

```
    0028 => "0000111010000000", -- 001C - 0E80                          test
r2
    0030 => "1000010000000010", -- 001E - 8402
brr.z         ResetExecute_1
    0032 => "1000000111110011", -- 0020 - 81F3                          brr
WaitForever
    0034 => "0010010100000100", -- 0022 - 2504
loadimm.upper BootVector_1.hi
    0036 => "0010010000000010", -- 0024 - 2402
loadimm.lower BootVector_1.lo
    0038 => "0010000111111000", -- 0026 - 21F8                          load
r7,r7
    0040 => "0000000000000000", -- 0028 - 0000                          nop
    0042 => "0000000000000000", -- 002A - 0000                          nop
    0044 => "0010010000000000", -- 002C - 2400
loadimm.lower 0x00
    0046 => "0010011010111000", -- 002E - 26B8                          mov
r2,r7
    0048 => "0010010100100100", -- 0030 - 2524
loadimm.upper 0x24
    0050 => "0010010000000000", -- 0032 - 2400
loadimm.lower 0x00
    0052 => "0000010010010111", -- 0034 - 0497                          sub
r2,r2,r7
    0054 => "0000111010000000", -- 0036 - 0E80                          test
r2
    0056 => "1000010000000010", -- 0038 - 8402
brr.z         ResetExecute_2
    0058 => "1000000111100110", -- 003A - 81E6                          brr
WaitForever
    0060 => "0010010100000100", -- 003C - 2504
loadimm.upper BootVector_2.hi
    0062 => "0010010000000100", -- 003E - 2404
loadimm.lower BootVector_2.lo
    0064 => "0010000010111000", -- 0040 - 20B8                          load
r2,r7
    0066 => "0000000000000000", -- 0042 - 0000                          nop
    0068 => "0000000000000000", -- 0044 - 0000                          nop
    0070 => "0010010110000111", -- 0046 - 2587
loadimm.upper 0x87
    0072 => "0010010011000000", -- 0048 - 24C0
loadimm.lower 0xc0
```

```
    0074 => "0000010010010111", -- 004A - 0497                    sub
r2,r2,r7
    0076 => "0000111010000000", -- 004C - 0E80                    test
r2
    0078 => "1000010000000010", -- 004E - 8402
brr.z        ResetExecute_3
    0080 => "1000000111011011", -- 0050 - 81DB                    brr
WaitForever
    0082 => "0010010100000100", -- 0052 - 2504
loadimm.upper BootVector.hi
    0084 => "0010010000000000", -- 0054 - 2400
loadimm.lower BootVector.lo
    0086 => "1000011111000000", -- 0056 - 87C0                    br
r7,0
    0088 => "0010010100000000", -- 0058 - 2500 ResetLoad:
loadimm.upper 0x00
    0090 => "0010010010000000", -- 005A - 2480
loadimm.lower 0x80
    0092 => "0010011110111000", -- 005C - 27B8                    mov
r6,r7
    0094 => "0100001010000000", -- 005E - 4280 WaitFor_AA:        in
r2
    0096 => "0000100010010110", -- 0060 - 0896                    nand
r2,r2,r6
    0098 => "0000100010010010", -- 0062 - 0892                    nand
r2,r2,r2
    0100 => "0000111010000000", -- 0064 - 0E80                    test
r2
    0102 => "1000010111111100", -- 0066 - 85FC
brr.z        WaitFor_AA
    0104 => "0010010111111111", -- 0068 - 25FF
loadimm.upper 0xFF
    0106 => "0010010000000000", -- 006A - 2400
loadimm.lower 0x00
    0108 => "0100001010000000", -- 006C - 4280                    in
r2
    0110 => "0000100010010111", -- 006E - 0897                    nand
r2,r2,r7
    0112 => "0000100010010010", -- 0070 - 0892                    nand
r2,r2,r2
    0114 => "0010010110101010", -- 0072 - 25AA
LOADIMM.UPPER 0xAA
```

```
    0116 => "0000010010010111", -- 0074 - 0497                          sub
r2,r2,r7
    0118 => "0000111010000000", -- 0076 - 0E80                          test
r2
    0120 => "1000010000000010", -- 0078 - 8402
brr.z         Got_AA
    0122 => "1000000111110010", -- 007A - 81F2                          brr
WaitFor_AA
    0124 => "0010010000000001", -- 007C - 2401 Got_AA:
loadimm.lower 0x01
    0126 => "0000000000000000", -- 007E - 0000                          nop
    0128 => "0000000000000000", -- 0080 - 0000                          nop
    0130 => "0000000000000000", -- 0082 - 0000                          nop
    0132 => "0100000111000000", -- 0084 - 41C0                          out
r7
    0134 => "0100001010000000", -- 0086 - 4280                          in
r2
    0136 => "0000100010010110", -- 0088 - 0896                          nand
r2,r2,r6
    0138 => "0000100010010010", -- 008A - 0892                          nand
r2,r2,r2
    0140 => "0000111010000000", -- 008C - 0E80                          test
r2
    0142 => "1000010000000010", -- 008E - 8402
brr.z         Done_AA
    0144 => "1000000111111011", -- 0090 - 81FB                          brr
WaitForEnd_AA
    0146 => "0010010000000000", -- 0092 - 2400 Done_AA:
loadimm.lower 0x00
    0148 => "0000000000000000", -- 0094 - 0000                          nop
    0150 => "0000000000000000", -- 0096 - 0000                          nop
    0152 => "0000000000000000", -- 0098 - 0000                          nop
    0154 => "0100000111000000", -- 009A - 41C0                          out
r7
    0156 => "0010010100000100", -- 009C - 2504
loadimm.upper RamStart.hi
    0158 => "0010010000000000", -- 009E - 2400
loadimm.lower RamStart.lo
    0160 => "0010011110111000", -- 00A0 - 27B8                          mov
r6,r7
    0162 => "0010010100000000", -- 00A2 - 2500
loadimm.upper 0x00
```

```
    0164 => "0010010000000010", -- 00A4 - 2402
loadimm.lower 0x02
    0166 => "0010001110111000", -- 00A6 - 23B8
store          r6,r7
    0168 => "0010010100000000", -- 00A8 - 2500
loadimm.upper 0x00
    0170 => "0010010010000000", -- 00AA - 2480
loadimm.lower 0x80
    0172 => "0010011110111000", -- 00AC - 27B8                       mov
r6,r7
    0174 => "0100001010000000", -- 00AE - 4280 WaitFor_55:          in
r2
    0176 => "0000100010010110", -- 00B0 - 0896                       nand
r2,r2,r6
    0178 => "0000100010010010", -- 00B2 - 0892                       nand
r2,r2,r2
    0180 => "0000111010000000", -- 00B4 - 0E80                       test
r2
    0182 => "1000010111111100", -- 00B6 - 85FC
brr.z          WaitFor_55
    0184 => "0010010111111111", -- 00B8 - 25FF
loadimm.upper 0xFF
    0186 => "0010010000000000", -- 00BA - 2400
loadimm.lower 0x00
    0188 => "0100001010000000", -- 00BC - 4280                       in
r2
    0190 => "0000100010010111", -- 00BE - 0897                       nand
r2,r2,r7
    0192 => "0000100010010010", -- 00C0 - 0892                       nand
r2,r2,r2
    0194 => "0010010101010101", -- 00C2 - 2555
LOADIMM.UPPER 0x55
    0196 => "0000010010010111", -- 00C4 - 0497                       sub
r2,r2,r7
    0198 => "0000111010000000", -- 00C6 - 0E80                       test
r2
    0200 => "1000010000000010", -- 00C8 - 8402
brr.z          Got_55
    0202 => "1000000111110010", -- 00CA - 81F2                       brr
WaitFor_55
    0204 => "0010010000000001", -- 00CC - 2401 Got_55:
loadimm.lower 0x01
    0206 => "0000000000000000", -- 00CE - 0000                       nop
    0208 => "0000000000000000", -- 00D0 - 0000                    nop
    0210 => "0000000000000000", -- 00D2 - 0000                    nop
    0212 => "0100000111000000", -- 00D4 - 41C0                    out
r7
    0214 => "0100001010000000", -- 00D6 - 4280                    in
r2
    0216 => "0000100010010110", -- 00D8 - 0896                    nand
r2,r2,r6
    0218 => "0000100010010010", -- 00DA - 0892                    nand
r2,r2,r2
    0220 => "0000111010000000", -- 00DC - 0E80                    test
r2
    0222 => "1000010000000010", -- 00DE - 8402
brr.z         Done_55
    0224 => "1000000111111011", -- 00E0 - 81FB                    brr
WaitForEnd_55
    0226 => "0010010000000000", -- 00E2 - 2400 Done_55:
loadimm.lower 0x00
    0228 => "0000000000000000", -- 00E4 - 0000                    nop
    0230 => "0000000000000000", -- 00E6 - 0000                    nop
    0232 => "0000000000000000", -- 00E8 - 0000                    nop
    0234 => "0100000111000000", -- 00EA - 41C0                    out
r7
    0236 => "0010010100000000", -- 00EC - 2500
loadimm.upper 0x00
    0238 => "0010010010000000", -- 00EE - 2480
loadimm.lower 0x80
    0240 => "0010011110111000", -- 00F0 - 27B8                    mov
r6,r7
    0242 => "0100001010000000", -- 00F2 - 4280 WaitForSize:       in
r2
    0244 => "0010011100010000", -- 00F4 - 2710                    mov
r4,r2
    0246 => "0000100010010110", -- 00F6 - 0896                    nand
r2,r2,r6
    0248 => "0000100010010010", -- 00F8 - 0892                    nand
r2,r2,r2
    0250 => "0000111010000000", -- 00FA - 0E80                    test
r2
    0252 => "1000010111111011", -- 00FC - 85FB
brr.z         WaitForSize
    0254 => "0000110100001000", -- 00FE - 0D08                    shr
r4,8
```

```
    0256 => "0010010000000001", -- 0100 - 2401
loadimm.lower 0x01
    0258 => "0000000000000000", -- 0102 - 0000                          nop
    0260 => "0000000000000000", -- 0104 - 0000                          nop
    0262 => "0000000000000000", -- 0106 - 0000                          nop
    0264 => "0100000111000000", -- 0108 - 41C0                          out
r7
    0266 => "0100001010000000", -- 010A - 4280                          in
r2
    0268 => "0000100010010110", -- 010C - 0896                          nand
r2,r2,r6
    0270 => "0000100010010010", -- 010E - 0892                          nand
r2,r2,r2
    0272 => "0000111010000000", -- 0110 - 0E80                          test
r2
    0274 => "1000010000000010", -- 0112 - 8402
brr.z          DoneSize
    0276 => "1000000111111011", -- 0114 - 81FB                          brr
WaitForSizeEnd
    0278 => "0010010000000000", -- 0116 - 2400 DoneSize:
loadimm.lower 0x00
    0280 => "0000000000000000", -- 0118 - 0000                          nop
    0282 => "0000000000000000", -- 011A - 0000                          nop
    0284 => "0000000000000000", -- 011C - 0000                          nop
    0286 => "0100000111000000", -- 011E - 41C0                          out
r7
    0288 => "0010010100000010", -- 0120 - 2502
loadimm.upper 0x02
    0290 => "0010010000000000", -- 0122 - 2400
loadimm.lower 0x00
    0292 => "0010011011111000", -- 0124 - 26F8                          mov
r3,r7
    0294 => "0010010111111111", -- 0126 - 25FF GetProgram:
loadimm.upper LedDisplay.hi
    0296 => "0010010011110010", -- 0128 - 24F2
loadimm.lower LedDisplay.lo
    0298 => "0000000000000000", -- 012A - 0000                          nop
    0300 => "0000000000000000", -- 012C - 0000                          nop
    0302 => "0000000000000000", -- 012E - 0000                          nop
    0304 => "0000000000000000", -- 0130 - 0000                          nop
    0306 => "0000000000000000", -- 0132 - 0000                          nop
    0308 => "0010001111100000", -- 0134 - 23E0
store          r7,r4
```

```
    0310 => "0000111100000000", -- 0136 - 0F00                        test
r4
    0312 => "1000010101100111", -- 0138 - 8567
brr.z         WaitForever
    0314 => "0100001010000000", -- 013A - 4280                        in
r2
    0316 => "0010011001010000", -- 013C - 2650                        mov
r1,r2
    0318 => "0000100010010110", -- 013E - 0896                        nand
r2,r2,r6
    0320 => "0000100010010010", -- 0140 - 0892                        nand
r2,r2,r2
    0322 => "0000111010000000", -- 0142 - 0E80                        test
r2
    0324 => "1000010111111011", -- 0144 - 85FB
brr.z         WaitForHighByte
    0326 => "0000110001001000", -- 0146 - 0C48                        shr
r1,8
    0328 => "0000101001001000", -- 0148 - 0A48                        shl
r1,8
    0330 => "0010010000000001", -- 014A - 2401
loadimm.lower 0x01
    0332 => "0000000000000000", -- 014C - 0000                        nop
    0334 => "0000000000000000", -- 014E - 0000                        nop
    0336 => "0000000000000000", -- 0150 - 0000                        nop
    0338 => "0100000111000000", -- 0152 - 41C0                        out
r7
    0340 => "0100001010000000", -- 0154 - 4280                        in
r2
    0342 => "0000100010010110", -- 0156 - 0896                        nand
r2,r2,r6
    0344 => "0000100010010010", -- 0158 - 0892                        nand
r2,r2,r2
    0346 => "0000111010000000", -- 015A - 0E80                        test
r2
    0348 => "1000010000000010", -- 015C - 8402
brr.z         DoneHighByte
    0350 => "1000000111111011", -- 015E - 81FB                        brr
WaitForHighByteEnd
    0352 => "0010010000000000", -- 0160 - 2400
loadimm.lower 0x00
    0354 => "0000000000000000", -- 0162 - 0000                        nop
    0356 => "0000000000000000", -- 0164 - 0000                        nop
    0358 => "0000000000000000", -- 0166 - 0000                      nop
    0360 => "0100000111000000", -- 0168 - 41C0                      out
r7
    0362 => "0010011000001000", -- 016A - 2608                      mov
r0,r1
    0364 => "0100001010000000", -- 016C - 4280                      in
r2
    0366 => "0010011001010000", -- 016E - 2650                      mov
r1,r2
    0368 => "0000100010010110", -- 0170 - 0896                      nand
r2,r2,r6
    0370 => "0000100010010010", -- 0172 - 0892                      nand
r2,r2,r2
    0372 => "0000111010000000", -- 0174 - 0E80                      test
r2
    0374 => "1000010111111011", -- 0176 - 85FB
brr.z         WaitForLowByte
    0376 => "0000110001001000", -- 0178 - 0C48                      shr
r1,8
    0378 => "0010010000000001", -- 017A - 2401
loadimm.lower 0x01
    0380 => "0000000000000000", -- 017C - 0000                      nop
    0382 => "0000000000000000", -- 017E - 0000                      nop
    0384 => "0000000000000000", -- 0180 - 0000                      nop
    0386 => "0100000111000000", -- 0182 - 41C0                      out
r7
    0388 => "0100001010000000", -- 0184 - 4280                      in
r2
    0390 => "0000100010010110", -- 0186 - 0896                      nand
r2,r2,r6
    0392 => "0000100010010010", -- 0188 - 0892                      nand
r2,r2,r2
    0394 => "0000111010000000", -- 018A - 0E80                      test
r2
    0396 => "1000010000000010", -- 018C - 8402
brr.z         DoneLowByte
    0398 => "1000000111111011", -- 018E - 81FB                      brr
WaitForLowByteEnd
    0400 => "0100001010000000", -- 0190 - 4280 DoneLowByte:         in
r2
    0402 => "0010010000000000", -- 0192 - 2400
loadimm.lower 0x00
    0404 => "0000000000000000", -- 0194 - 0000                      nop
    0406 => "0000000000000000", -- 0196 - 0000                                   nop
    0408 => "0000000000000000", -- 0198 - 0000                                   nop
    0410 => "0100000111000000", -- 019A - 41C0                                   out
r7
    0412 => "0000001001001000", -- 019C - 0248                                   add
r1,r1,r0
    0414 => "0000101010001001", -- 019E - 0A89                                   shl
r2,9
    0416 => "0000110010001111", -- 01A0 - 0C8F                                   shr
r2,15
    0418 => "0000111010000000", -- 01A2 - 0E80                                   test
r2
    0420 => "1000010000010101", -- 01A4 - 8415
brr.z         GotInstruction
    0422 => "0010011011001000", -- 01A6 - 26C8                                   mov
r3,r1
    0424 => "0010010100000000", -- 01A8 - 2500
loadimm.upper StepSize.hi
    0426 => "0010010000000010", -- 01AA - 2402
loadimm.lower StepSize.lo
    0428 => "0010011010111000", -- 01AC - 26B8                                   mov
r2,r7
    0430 => "0010010100000100", -- 01AE - 2504
loadimm.upper BootVector.hi
    0432 => "0010010000000000", -- 01B0 - 2400
loadimm.lower BootVector.lo
    0434 => "0010011000111000", -- 01B2 - 2638                                   mov
r0,r7
    0436 => "0010011111011000", -- 01B4 - 27D8                                   mov
r7,r3
    0438 => "0000110111001000", -- 01B6 - 0DC8                                   shr
r7,8
    0440 => "0010010100100101", -- 01B8 - 2525
loadimm.upper 0x25
    0442 => "0010001000111000", -- 01BA - 2238
store         r0,r7
    0444 => "0000001000000010", -- 01BC - 0202                                   add
r0,r0,r2
    0446 => "0010011111011000", -- 01BE - 27D8                                   mov
r7,r3
    0448 => "0010010100100100", -- 01C0 - 2524
loadimm.upper 0x24
```

```
    0450 => "0010001000111000", -- 01C2 - 2238
store         r0,r7
    0452 => "0000001000000010", -- 01C4 - 0202                     add
r0,r0,r2
    0454 => "0010010110000111", -- 01C6 - 2587
loadimm.upper 0x87
    0456 => "0010010011000000", -- 01C8 - 24C0
loadimm.lower 0xC0
    0458 => "0010001000111000", -- 01CA - 2238
store         r0,r7
    0460 => "1000000000000101", -- 01CC - 8005                     brr
DecrementCount
    0462 => "0010001011001000", -- 01CE - 22C8 GotInstruction:
store         r3,r1
    0464 => "0010010100000000", -- 01D0 - 2500
loadimm.upper StepSize.hi
    0466 => "0010010000000010", -- 01D2 - 2402
loadimm.lower StepSize.lo
    0468 => "0000001011011111", -- 01D4 - 02DF                     add
r3,r3,r7
    0470 => "0010010100000000", -- 01D6 - 2500
loadimm.upper 0x00
    0472 => "0010010000000001", -- 01D8 - 2401
loadimm.lower 0x01
    0474 => "0000010100100111", -- 01DA - 0527                     sub
r4,r4,r7
    0476 => "1000000110100101", -- 01DC - 81A5                     brr
GetProgram
    others => (others => '0')
);

begin
    process(clk)
    begin
        if not rising_edge(clk) then

            if en = '1' then
                dout <= mem(to_integer(unsigned(addr(7 downto 0))));
            else
                dout <= (others => '0');
            end if;

        end if;
end process;

end Behavioral;
```

EXMEM_latches.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity EXMEM_latches is
    port(
        -- CONTROL SIGNALS
        clk,en,rst : in std_logic;

        -- INPUTS
        PC_in : in std_logic_vector(15 downto 0);
        PC_tb_in : in std_logic_vector(15 downto 0);
        instr_in : in std_logic_vector(15 downto 0);

        mem_wr_en_in : in std_logic;
        mem_rd_en_in : in std_logic;


        wb_en_in : in std_logic;
        wb_idx_in : in std_logic_vector(2 downto 0);

        din_in : in std_logic_vector(15 downto 0);
        result_in : in std_logic_vector(15 downto 0);

        -- OUTPUTS
        PC_out : out std_logic_vector(15 downto 0);
        PC_tb_out : out std_logic_vector(15 downto 0);
        instr_out : out std_logic_vector(15 downto 0);

        mem_wr_en_out : out std_logic;
        mem_rd_en_out : out std_logic;

        wb_en_out : out std_logic;
        wb_idx_out : out std_logic_vector(2 downto 0);
        din_out : out std_logic_vector(15 downto 0);
        result_out : out std_logic_vector(15 downto 0)

    );
end EXMEM_latches;

architecture Behavioral of EXMEM_latches is

signal PC : std_logic_vector(15 downto 0) := x"0000";

signal PC_tb : std_logic_vector(15 downto 0) := x"FFFF";
signal instr : std_logic_vector(15 downto 0) := x"0000";

signal mem_wr_en : std_logic := '0';
signal mem_rd_en : std_logic := '0';
signal wb_en : std_logic := '0';
signal wb_idx : std_logic_vector(2 downto 0);
signal din : std_logic_vector(15 downto 0);
signal result : std_logic_vector(15 downto 0);


begin

    process(clk)
    begin
    if rising_edge(clk) then
        if (rst = '1') then
--            PC <= x"0000";
            PC_TB <= x"FFFF";
            instr <= (others => '0');
            mem_wr_en <= '0';
            mem_rd_en <= '0';
            wb_en <= '0';
            wb_idx <= (others => '0');
            din <= (others => '0');
            result <= (others => '0');

        elsif (en = '1') then
            PC <= PC_in;
            PC_TB <= PC_tb_in;
            instr <= instr_in;
            mem_wr_en <= mem_wr_en_in;
            mem_rd_en <= mem_rd_en_in;
            wb_en <= wb_en_in;
            wb_idx <= wb_idx_in;
            din <= din_in;
            result <= result_in;

        end if;
    end if;
    end process;

    PC_out <= PC;
    PC_tb_out <= PC_tb;
    instr_out <= instr;
    mem_wr_en_out <= mem_wr_en;
    mem_rd_en_out <= mem_rd_en;
    wb_en_out <= wb_en;
    wb_idx_out <= wb_idx;
    din_out <= din;
    result_out <= result;


end Behavioral;
```

IDEX_latches.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
--use ieee.std_logic_unsigned.all;

entity IDEX_latches is
    port(
        -- CONTROL SIGNALS
        clk,en,rst : in std_logic;

        -- INPUTS
        pc_in : in std_logic_vector(15 downto 0);
        instr_in : in std_logic_vector(15 downto 0);
        opcode_in : in std_logic_vector(6 downto 0);
        out_flag_in : in std_logic;

        in1_in,in2_in : in std_logic_vector(15 downto 0);

        mem_wr_en_in : in std_logic;
        mem_rd_en_in : in std_logic;


        wb_en_in : in std_logic;
        wb_idx_in : in std_logic_vector(2 downto 0);

        alu_mode_in : in std_logic_vector(2 downto 0);


        -- OUTPUTS
        pc_out : out std_logic_vector(15 downto 0);
        instr_out : out std_logic_vector(15 downto 0);
        opcode_out : out std_logic_vector(6 downto 0);
        out_flag_out : out std_logic;

        in1_out,in2_out : out std_logic_vector(15 downto 0);

        mem_wr_en_out : out std_logic;
        mem_rd_en_out : out std_logic;


        wb_en_out : out std_logic;
        wb_idx_out : out std_logic_vector(2 downto 0);

        alu_mode_out : out std_logic_vector(2 downto 0)
    );
end IDEX_latches;

architecture Behavioral of IDEX_latches is

signal instr : STD_LOGIC_VECTOR(15 downto 0) := (others => '0');
signal PC : STD_LOGIC_VECTOR(15 downto 0) := x"FFFF";
signal opcode : std_logic_vector(6 downto 0);
signal out_flag : std_logic := '0';
signal in1, in2 : std_logic_vector(15 downto 0);
signal mem_wr_en : std_logic := '0';
signal mem_rd_en : std_logic := '0';
signal wb_en : std_logic := '0';
signal wb_idx : std_logic_vector(2 downto 0);
signal alu_mode : std_logic_vector(2 downto 0);

begin

    process(clk)
    begin
    if rising_edge(clk) then
        if (rst = '1') then
            opcode <= (others => '0');
            out_flag <= '0';

            instr <= (others => '0');
            pc <= x"FFFF";

            in1 <= (others => '0');
            in2 <= (others => '0');

            mem_wr_en <= '0';
            mem_rd_en <= '0';

            wb_en <= '0';
            wb_idx <= (others => '0');

            alu_mode <= (others => '0');

        elsif en = '1' then
            opcode <= opcode_in;
            out_flag <= out_flag_in;

            pc <= pc_in;
            instr <= instr_in;

            in1 <= in1_in;
            in2 <= in2_in;

            mem_wr_en <= mem_wr_en_in;
            mem_rd_en <= mem_rd_en_in;

            wb_en <= wb_en_in;
            wb_idx <= wb_idx_in;

            alu_mode <= alu_mode_in;
        end if;
    end if;
    end process;

    opcode_out <= opcode;
    out_flag_out <= out_flag;

    pc_out <= pc;
    instr_out <= instr;

    in1_out <= in1;
    in2_out <= in2;

    mem_wr_en_out <= mem_wr_en;
    mem_rd_en_out <= mem_rd_en;

    wb_en_out <= wb_en;
    wb_idx_out <= wb_idx;

    alu_mode_out <= alu_mode;


end Behavioral ; -- Behavioral
```

IFID_latches.vhd

```


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;



-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;
```

```
entity IFID_latches is
    Port (

        -- CONTROL SIGNALS
        clk,en,rst : in std_logic;

        PC_in : in STD_LOGIC_VECTOR(15 downto 0);
        PC_out : out STD_LOGIC_VECTOR(15 downto 0);

        instr_in : in STD_LOGIC_VECTOR(15 downto 0);
        instr_out : out STD_LOGIC_VECTOR(15 downto 0)
    );

end IFID_latches;

architecture Behavioral of IFID_latches is

signal instr : STD_LOGIC_VECTOR(15 downto 0) := (others => '0');
signal PC : STD_LOGIC_VECTOR(15 downto 0) := x"FFFF";

begin

    process(clk)
    begin
        if rising_edge(clk) then
            if (rst = '1') then
                instr <= (others => '0');
                PC <= x"FFFF";

            elsif en = '1' then
                instr <= instr_in;
                PC <= PC_in;
            end if;
        end if;

    end process;

    instr_out <= instr;
    PC_out <= PC;

end Behavioral;
```

Instr_decoder.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;


-- DESCRIPTION
-- Takes an instruction from the fetch register (+ CLK_in and EN)
-- Outputs an ALU mode code (3 bits), 3 3-bit register addresses


entity instr_decoder is
port (
    -- INPUTS
    instr : in STD_LOGIC_VECTOR(15 downto 0);
    in_port : in STD_LOGIC_VECTOR(15 downto 0);
    stall : in STD_LOGIC;

    -- OUTPUTS
    out_flag : out STD_LOGIC;

    rd_idx1 : out STD_LOGIC_VECTOR(2 downto 0);
    rd_idx2 : out STD_LOGIC_VECTOR(2 downto 0);

    imm_val : out STD_LOGIC_VECTOR(15 downto 0);
    imm_en : out STD_LOGIC;

    opcode : out STD_LOGIC_VECTOR(6 downto 0);
    alu_mode : out STD_LOGIC_VECTOR(2 downto 0);

    mem_wr_en : out STD_LOGIC;
    mem_rd_en : out STD_LOGIC;

    wb_en : out STD_LOGIC;
    wb_idx : out STD_LOGIC_VECTOR(2 downto 0);
    mask : out STD_LOGIC_VECTOR(15 downto 0)



);
end instr_decoder;

architecture Behavioral of instr_decoder is
begin

    process(instr, in_port) begin

--        if stall = '1' then
--            opcode <= (others => '0');
--            alu_mode <= (others => '0');

--            out_flag <= '0';

--            wb_en <= '0';
--            wb_idx <= (others => '0');

--            rd_idx1 <= (others => '0');
--            rd_idx2 <= (others => '0');

--            imm_val <= (others => '0');
--            imm_en <= '1';

--            mem_wr_en <= '0';
--            mem_rd_en <= '0';

--            mask <= x"0000";

--        else
            case to_integer(unsigned(instr(15 downto 9))) is

                when 0 =>
                    -- A0
                    opcode <= instr(15 downto 9);
                    alu_mode <= (others => '0');
```

```
        out_flag <= '0';

        wb_en <= '0';
        wb_idx <= (others => '0');

        rd_idx1 <= (others => '0');
        rd_idx2 <= (others => '0');

        imm_val <= (others => '0');
        imm_en <= '0';

        mem_wr_en <= '0';
        mem_rd_en <= '0';

        mask <= x"FFFF";

    when 1 | 2 | 3 =>
        -- A1
        opcode <= instr(15 downto 9);
        alu_mode <= instr(11 downto 9);

        out_flag <= '0';

        wb_en <= '1';
        wb_idx <= instr(8 downto 6);

        rd_idx1 <= instr(5 downto 3);
        rd_idx2 <= instr(2 downto 0);

        imm_val <= (others => '0');
        imm_en <= '0';

        mem_wr_en <= '0';
        mem_rd_en <= '0';

        mask <= x"FFFF";

    when 4 =>
        -- A1 (NAND CASE)
        opcode <= instr(15 downto 9);
        alu_mode <= instr(11 downto 9);
```

```
                    out_flag <= '0';

                    wb_en <= '1';
                    wb_idx <= instr(8 downto 6);

                    rd_idx1 <= instr(8 downto 6);
                    rd_idx2 <= instr(5 downto 3);

                    imm_val <= (others => '0');
                    imm_en <= '0';

                    mem_wr_en <= '0';
                    mem_rd_en <= '0';

                    mask <= x"FFFF";
                when 5 | 6 =>
                    -- A2
                    opcode <= instr(15 downto 9);
                    alu_mode <= instr(11 downto 9);

                    out_flag <= '0';

                    wb_en <= '1';
                    wb_idx <= instr(8 downto 6);

                    rd_idx1 <= instr(8 downto 6);
                    rd_idx2 <= (others => '0');

--                    imm_val(3 downto 0) <= instr(3 downto 0);
--                    imm_val(15 downto 4) <= (others => instr(3));
                    imm_val <= std_logic_vector(resize(signed(instr(3
downto 0)), 16));
                    imm_en <= '1';

                    mem_wr_en <= '0';
                    mem_rd_en <= '0';

                    mask <= x"FFFF";

                when 7 =>
                    --TEST
                    opcode <= instr(15 downto 9);
                    alu_mode <= instr(11 downto 9);
                    out_flag <= '0';

                    wb_en <= '0';
                    wb_idx <= (others => '0');

                    rd_idx1 <= instr(8 downto 6);
                    rd_idx2 <= (others => '0');

                    imm_val <= (others => '0');
                    imm_en <= '0';

                    mem_wr_en <= '0';
                    mem_rd_en <= '0';

                    mask <= x"FFFF";

                when 64 | 65 | 66 =>
                    -- B1
                    opcode <= instr(15 downto 9);
                    alu_mode <= (others => '0');

                    out_flag <= '0';

                    wb_en <= '0';
                    wb_idx <= (others => '0');

                    rd_idx1 <= (others => '0');
                    rd_idx2 <= (others => '0');

--                    imm_val(8 downto 0) <= instr(8 downto 0);
--                    imm_val(15 downto 9) <= (others => instr(8));
                    imm_val <= std_logic_vector(resize(signed(instr(8
downto 0)), 16));
                    imm_en <= '1';

                    mem_wr_en <= '0';
                    mem_rd_en <= '0';

                    mask <= x"FFFF";

                when 67 | 68 | 69 =>
                    -- B2
                    opcode <= instr(15 downto 9);
                    alu_mode <= (others => '0');

                    out_flag <= '0';

                    wb_en <= '0';
                    wb_idx <= (others => '0');

                    rd_idx1 <= instr(8 downto 6);
                    rd_idx2 <= (others => '0');

--                    imm_val(5 downto 0) <= instr(5 downto 0);
--                    imm_val(15 downto 6) <= (others => instr(5));
                    imm_val <= std_logic_vector(resize(signed(instr(5
downto 0)), 16));
                    imm_en <= '1';

                    mem_wr_en <= '0';
                    mem_rd_en <= '0';

                    mask <= x"FFFF";
                when 70 =>
                    -- BR.SUB
                    opcode <= instr(15 downto 9);
                    alu_mode <= (others => '0');

                    out_flag <= '0';

                    wb_en <= '1';
                    wb_idx <= "111";

                    rd_idx1 <= instr(8 downto 6);
                    rd_idx2 <= (others => '0');

--                    imm_val(5 downto 0) <= instr(5 downto 0);
--                    imm_val(15 downto 6) <= (others => instr(5));
                    imm_val <= std_logic_vector(resize(signed(instr(5
downto 0)), 16));
                    imm_en <= '1';

                    mem_wr_en <= '0';
                    mem_rd_en <= '0';
```

```
             mask <= x"FFFF";

         when 71 =>
             -- RETURN
             opcode <= instr(15 downto 9);
             alu_mode <= (others => '0');

             out_flag <= '0';

             wb_en <= '0';
             wb_idx <= (others => '0');

             rd_idx1 <= "111";
             rd_idx2 <= (others => '0');

             imm_val <= (others => '0');
             imm_en <= '0';

             mem_wr_en <= '0';
             mem_rd_en <= '0';

             mask <= x"FFFF";

        when 16 =>
             opcode <= instr(15 downto 9);
             alu_mode <= (others => '0');

             out_flag <= '0';

             wb_en <= '1';
             wb_idx <= instr(8 downto 6);

             rd_idx1 <= instr(5 downto 3);
             rd_idx2 <= (others => '0');

             imm_val <= (others => '0');
             imm_en <= '0';

             mem_wr_en <= '0';
             mem_rd_en <= '1';

             mask <= x"FFFF";
```

```
               when 17 =>
                    opcode <= instr(15 downto 9);
                    alu_mode <= (others => '0');

                    out_flag <= '0';

                    wb_en <= '0';
                    wb_idx <= (others => '0');

                    rd_idx1 <= instr(8 downto 6);
                    rd_idx2 <= instr(5 downto 3);

                    imm_val <= (others => '0');
                    imm_en <= '0';

                    mem_wr_en <= '1';
                    mem_rd_en <= '0';

                    mask <= x"FFFF";


               when 18 =>
               --LOADIMM
                   opcode <= instr(15 downto 9);
                   alu_mode <= "001"; -- Need to add the masked r7 to
the immediate value

                   out_flag <= '0';

                   wb_en <= '1'; -- Need to write back to register r7
                   wb_idx <= "111";

                   rd_idx1 <= "111";
                   rd_idx2 <= (others => '0');

                   imm_en <= '1';

                   mem_wr_en <= '0';
                   mem_rd_en <= '0';


                   -- Modulating imm val and mask based on changing
upper or lower bits of r7
```

```vhdl
               imm_val <= (others => '0');
               if (instr(8) = '1') then

                    mask <= x"00FF";
                    imm_val(15 downto 8) <= instr(7 downto 0);

               elsif (instr(8) = '0') then

                    mask <= x"FF00";
                    imm_val(7 downto 0) <= instr(7 downto 0);
               end if;





           when 19 =>
                     opcode <= instr(15 downto 9);
                     alu_mode <= (others => '0');

                     out_flag <= '0';

                     wb_en <= '1';
                     wb_idx <= instr(8 downto 6);

                     rd_idx1 <= instr(5 downto 3);
                     rd_idx2 <= (others => '0');

                     imm_val <= (others => '0');
                     imm_en <= '0';

                     mem_wr_en <= '0';
                     mem_rd_en <= '0';

                    mask <= x"FFFF";

           when 32 =>
--           OUT PORT
                    opcode <= (others => '0');
                    alu_mode <= (others => '0');

                    out_flag <= '1';
```

```
                        wb_en <= '0';
                        wb_idx <= (others => '0');

                        rd_idx1 <= instr(8 downto 6);
                        rd_idx2 <= (others => '0');

                        imm_val <= (others => '0');
                        imm_en <= '0';

                        mem_wr_en <= '0';
                        mem_rd_en <= '0';

                        mask <= x"FFFF";

               when 33 =>
    --           IN PORT
                        opcode <= (others => '0');
                        alu_mode <= "001";

                        out_flag <= '0';

                        wb_en <= '1';
                        wb_idx <= instr(8 downto 6);

                        rd_idx1 <= (others => '0');
                        rd_idx2 <= (others => '0');

                        imm_val <= in_port;
                        imm_en <= '1';

                        mem_wr_en <= '0';
                        mem_rd_en <= '0';

                        mask <= x"0000";

                when others =>
                    null;
            end case;
--        end if;
    end process;

end Behavioral;
```

MEMWB_latch.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity MEMWB_latches is
    port(
        -- CONTROL SIGNALS
        clk,en,rst : in std_logic;

        -- INPUTS
        pc_in,instr_in : in std_logic_vector(15 downto 0);
        wb_data_in : in std_logic_vector(15 downto 0);
        wb_en_in : in std_logic;
        wb_idx_in : in std_logic_vector(2 downto 0);

        -- OUTPUTS
        pc_out,instr_out : out std_logic_vector(15 downto 0);
        wb_data_out : out std_logic_vector(15 downto 0);
        wb_en_out : out std_logic;
        wb_idx_out : out std_logic_vector(2 downto 0)
    );
end MEMWB_latches;

architecture Behavioral of MEMWB_latches is

signal wb_data : std_logic_vector(15 downto 0);
signal wb_en : std_logic := '0';
signal wb_idx : std_logic_vector(2 downto 0);
signal PC : std_logic_vector(15 downto 0) := x"FFFF";
signal instr : std_logic_vector(15 downto 0) := x"0000";

begin

    process(clk)
    begin
    if rising_edge(clk) then
        if (rst = '1') then
```

```
            pc <= x"FFFF";
            instr <= (others => '0');
            wb_data <= (others => '0');
            wb_en <= '0';
            wb_idx <= (others => '0');
        elsif (en = '1') then
            pc <= pc_in;
            instr <= instr_in;
            wb_data <= wb_data_in;
            wb_en <= wb_en_in;
            wb_idx <= wb_idx_in;
        end if;
    end if;
    end process;

    pc_out <= pc;
    instr_out <= instr;
    wb_data_out <= wb_data;
    wb_en_out <= wb_en;
    wb_idx_out <= wb_idx;


end Behavioral;
```

Mem_sel.vhd

```
------------------------------------------------------------------------
-----------
-- Company:
-- Engineer:
--
-- Create Date: 03/24/2025 05:38:19 PM
-- Design Name:
-- Module Name: mem_sel - Behavioral
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
--
-- Dependencies:
```

```vhdl
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
-----------------------------------------------------------------------
-----------


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity mem_sel is
    Port ( pc : in STD_LOGIC_VECTOR (15 downto 0);
           mem_select : out STD_LOGIC;
           addr : out STD_LOGIC_VECTOR (15 downto 0));
end mem_sel;

architecture Behavioral of mem_sel is

begin
    process(pc) begin
        mem_select <= pc(11);

        addr <= (others => '0');
        addr(10 downto 0) <= pc(10 downto 0);
    end process;
end Behavioral;
```

Mux.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity mux is
    port(
    en    : in std_logic;
    in0   : in std_logic_vector(15 downto 0);
    in1   : in std_logic_vector(15 downto 0);
    output   : out std_logic_vector(15 downto 0)
);
end mux;

architecture Behavioral of mux is

begin
    process(en, in0, in1)
    begin
        if en = '1' then
            output <= in1;
        else
            output <= in0;
        end if;

    end process;

end Behavioral;
```

Register_file.vhd

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use IEEE.numeric_std.all; -- use that, it's a better coding guideline
entity register_file is
    port(

        -- Control Signals
        rst,clk: in std_logic;

        -- Reading signals
        rd_index1,rd_index2 : in std_logic_vector(2 downto 0);
        rd_data1,rd_data2: out std_logic_vector(15 downto 0);

        --Writing signals
        wr_index: in std_logic_vector(2 downto 0);
        wr_data: in std_logic_vector(15 downto 0);
        wr_en: in std_logic;

        r0,r1,r2,r3,r4,r5,r6,r7: out std_logic_vector(15 downto 0)
  );
end register_file;

architecture behavioural of register_file is
    type reg_array is array (integer range 0 to 7) of
std_logic_vector(15 downto 0);


signal reg_file : reg_array := (
        0 => x"0000",
        1 => x"0001",
        2 => x"0002",
        3 => x"0003",
        4 => x"0004",
        5 => x"0005",
        6 => x"0006", --IMM storage
        7 => x"0007" --PC storage
    );
```

```
begin

    process(clk, rst)
    begin
        if rst = '1' then
            for i in 0 to 7 loop
                reg_file(i) <= (others => '0');
            end loop;

        elsif not rising_edge(clk) and wr_en = '1' then
            reg_file(to_integer(unsigned(wr_index))) <= wr_data;
        end if;

    end process;

    rd_data1 <= reg_file(to_integer(unsigned(rd_index1)));
    rd_data2 <= reg_file(to_integer(unsigned(rd_index2)));

    r0 <= reg_file(0);
    r1 <= reg_file(1);
    r2 <= reg_file(2);
    r3 <= reg_file(3);
    r4 <= reg_file(4);
    r5 <= reg_file(5);
    r6 <= reg_file(6);
    r7 <= reg_file(7);

end behavioural;
```