NYU Processor Design Team SoC CPU Core

Michael Lippe, Electrical and Computer Engineering, Spring 2024

***Abstract*—The NYU Processor Design Team CPU is a single-core 32-bit design capable of executing the RISC-V 32I ISA.**

***Index Terms*—SoC, Processor Design, RISC-V, CPU, NYU Processor Design Team, NYU VIP**

1. INTRODUCTION

The NYU Processor Design VIP Team was founded in Spring 2023 with the goal of developing and synthesizing novel microprocessor designs. The team's inaugural project is a 32-bit SoC with a single-core CPU capable of executing the unprivileged version of the RISC-V 32I ISA.

Since an SoC consists of more than just a CPU, hardware development of the SoC is split across three sub-teams, Core, AMBA, and Memory, with each sub-team having its own leader, referred to as Czars, who is responsible for that sub-team's portion of the technology stack. The overall team operates on an open-source model with each sub-team having a main GitHub repository through which contributions are made. As Core Czar, this report will focus on the development of the SoC's CPU, though other components will be mentioned in passing as needed.

For the Core team, all hardware components are implemented using SystemVerilog and tested using Catch2. The SystemVerilog modules are converted to C++ objects with Verilator and CMake is used to configure and build the project for C++ testing. All core-related work is stored in the NYU Processor Design Team's nyu-core GitHub repository.

2. ANALYSIS OF APPLICABLE STANDARDS

The unprivileged version of the RISC-V 32I ISA consists of the instructions shown in Figure 1.

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]			000	rd	1100111	JALR
imm[12|10:5]		rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]		rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]		rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]		rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]		rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]		rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]			rs1	000	rd	0000011	LB
imm[11:0]			rs1	001	rd	0000011	LH
imm[11:0]			rs1	010	rd	0000011	LW
imm[11:0]			rs1	100	rd	0000011	LBU
imm[11:0]			rs1	101	rd	0000011	LHU
imm[11:5]		rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]		rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]		rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]			rs1	000	rd	0010011	ADDI
imm[11:0]			rs1	010	rd	0010011	SLTI
imm[11:0]			rs1	011	rd	0010011	SLTIU
imm[11:0]			rs1	100	rd	0010011	XORI
imm[11:0]			rs1	110	rd	0010011	ORI
imm[11:0]			rs1	111	rd	0010011	ANDI
0000000		shamt	rs1	001	rd	0010011	SLLI
0000000		shamt	rs1	101	rd	0010011	SRLI
0100000		shamt	rs1	101	rd	0010011	SRAI
0000000		rs2	rs1	000	rd	0110011	ADD
0100000		rs2	rs1	000	rd	0110011	SUB
0000000		rs2	rs1	001	rd	0110011	SLL
0000000		rs2	rs1	010	rd	0110011	SLT
0000000		rs2	rs1	011	rd	0110011	SLTU
0000000		rs2	rs1	100	rd	0110011	XOR
0000000		rs2	rs1	101	rd	0110011	SRL
0100000		rs2	rs1	101	rd	0110011	SRA
0000000		rs2	rs1	110	rd	0110011	OR
0000000		rs2	rs1	111	rd	0110011	AND
fm	pred | succ		rs1	000	rd	0001111	FENCE
000000000000			00000	000	00000	1110011	ECALL
000000000001			00000	000	00000	1110011	EBREAK

Figure 1: RISC-V 32I Instructions [1]

Since the I-type instructions are rather varied in their functionality, they were split into three types for ease of decoding. I type 1 instructions are the immediate operation instructions such as ADDI and XORI, I type 2 instructions are the load instructions, and the sole I type 3 instruction is the JALR instruction.

Since the CPU design only consists of one core, FENCE is treated as a NOP. The environment instructions, ECALL and EBREAK, are also treated as NOPs since their exact implementation is left up to the CPU designer to decide and they use the Control and Status Registers (CSRs) which are left

unimplemented in the current design since we are only interested in the unprivileged ISA.

3. DESIGN WORK

3.1. PIPELINE AND LATCHES

The block diagram for the core design is shown in Figure 2.

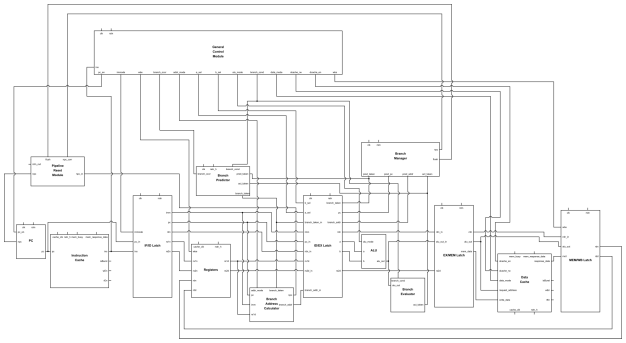

Figure 2: CPU Block Diagram

Figure 2 shows the modules that make up the core, which are as follows: Program Counter (PC), Instruction Cache, IF/ID Latch, CPU Registers, Branch Address Calculator, ID/EX Latch, Arithmetic Logic Unit (ALU), Branch Evaluator, EX/MEM Latch, Data Cache, MEM/WB Latch, Branch Manager, Branch Predictor, Pipeline Reset Module, and General Control Module.

To allow for faster clock speeds and a more modular design, the general design for the Core was chosen to be a multi-cycle pipelined design. To keep the general design similar to what members of the team would be familiar with from the Computer Architecture course, a five-stage pipeline was used with the following stages: Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Memory Access (MEM), Write Back (WB).

The names of the pipeline stages succinctly describe what happens in them. In the IF stage, an instruction is loaded from memory via the instruction cache based on the address stored in the program counter. In the ID stage, the previously fetched instruction is decoded to determine the relevant registers, immediate value, and potential branch address. In the EX stage, the logic calculation needed for the instruction is performed by the ALU. In the MEM stage, if an instruction accesses memory, the relevant address in memory is written to or read from via the data cache. In the WB stage, any relevant data is written to the specified destination register.

The modules that make up the CPU can generally be categorized by the pipeline stage or stages they belong to. Figure 3 shows the block diagram of the core color-coded by pipeline stage.

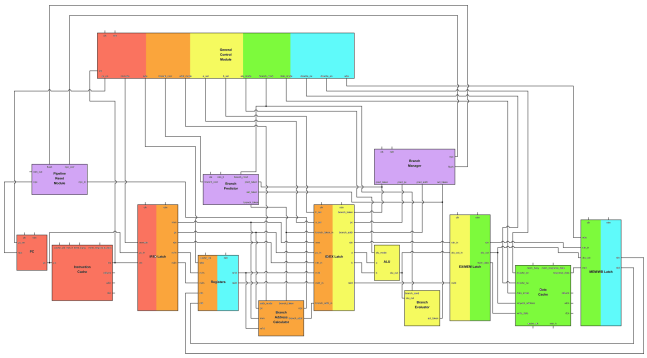

Figure 3: CPU Block Diagram Pipeline Stages

In Figure 3, red is for the IF stage, orange is for the ID stage, yellow is for the EX stage, green is for the MEM stage, and blue is for the WB stage. Modules with multiple colors can be considered part of two or more pipeline stages while modules colored purple, all of which are for branch prediction, do not neatly fit into any stage or set of stages.

To carry over and convert necessary values from one pipeline stage to the next, the four latch modules shown in Figure 4 were designed.

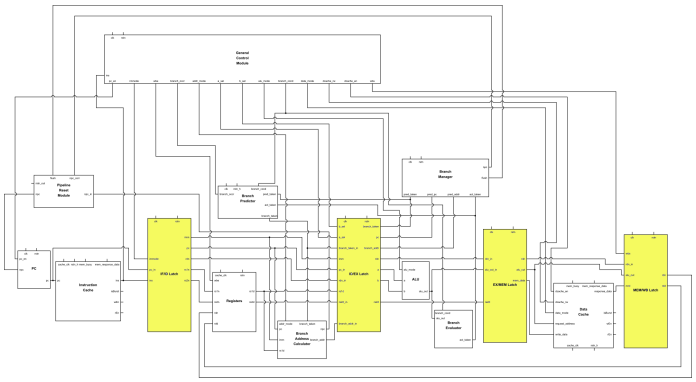

Figure 4: CPU Pipeline Latches

As shown in Figure 4, a latch module is present between the IF and ID stages, the ID and EX stages, the EX and MEM stages, and the MEM and WB stages, for a total of four latches.

The IF/ID latch carries over the PC value from the IF stage to the ID stage, determines the first, second, and destination registers from the instruction code, and determines the immediate value from the instruction based on the immode control signal. The immode control signal ranges in value from 0 to 5, with each value corresponding to a different method of constructing the immediate value from the 4-byte instruction. Table 1 shows the different immediate value construction methods and their corresponding immode value.

Table 1: immode Control Signal Function

immode	**Immediate Value (imm)**
0	32'b0
1	Sign Extend ins[31:20]
2	Sign Extend {ins[31:25], ins[11:7]}
3	Sign Extend {ins[31], ins[7], ins[30:25], ins[11:8], 1'b0}
4	{ins[31:12], 12'b0}
5	{11'b0, ins[31], ins[19:12], ins[20], ins[30:21], 1'b0}

The ID/EX latch passes the destination register number, the branch_taken signal from the branch predictor, the data from the second register, the calculated branch address, and the program counter value to the EX stage and determines the operands for the ALU based on the input signals and the a_sel and b_sel control signals. Tables 2 and 3 show the values that can be selected as an operand and their corresponding control signal value for operands a and b respectively.

Table 2: a_sel Control Signal Function

a_sel	**ALU Operand a (a)**
0	First register data
1	Passed in PC value
3	0

Table 3: b_sel Control Signal Function

b_sel	**ALU Operand b (b)**
0	Second register data
1	Immediate value
2	4
3	Immediate value << 12

The EX/MEM latch passes the destination register number and the ALU output data from the EX stage to the MEM stage, and passes in the data from the second register to be used as the data to write to memory.

Finally, the MEM/WB latch passes the destination register number from the MEM stage to the WB stage and calculates the data to store in the destination register based on the input signals and the wbs control signal. The wbs control signal ranges from 0 to 5 and Table 4 shows the different values that can be written to the destination register based on the wbs control signal value. Here, mrd is the data read from memory and alu_out is the ALU output.

Table 4: wbs Control Signal Function

wbs	**Destination Register Write Data (rdd)**
0	Sign Extend mrd[7:0]
1	Sign Extend mrd[15:0]
2	mrd
3	alu_out
4	Zero Extend mrd[7:0]
5	Zero Extend mrd[15:0]

3.2. INSTRUCTION FETCH STAGE

The IF stage consists of the Program Counter and Instruction Cache modules.

The PC module, shown in Figure 5, is a simple 32-bit register that receives input from the system responsible for branch prediction and can be enabled or disabled with the pc_en control signal.

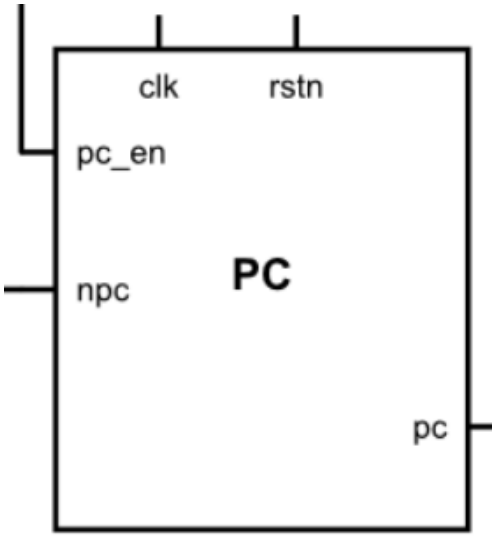

Figure 5: Program Counter Module

The Instruction Cache, shown in Figure 6, is a connection module containing the Instruction Cache Manager and L1 Instruction Cache modules.

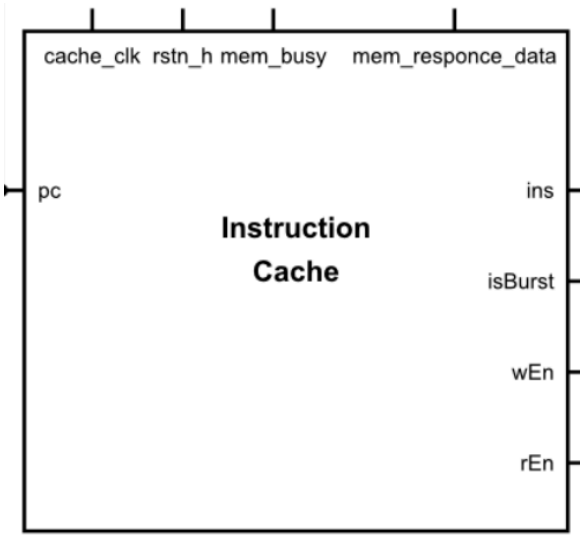

Figure 6: Instruction Cache Module

The Instruction Cache Manager module acts as a translation layer between the CPU, memory, and the L1 Instruction Cache. Its main function is to convert memory requests from the L1 Instruction Cache into the format accepted by AMBA which serves as the SoC's memory controller.

The L1 Instruction Cache module is still in development and its exact details have yet to be finalized.

3.3. INSTRUCTION DECODE STAGE

The ID stage consists of the General Purpose CPU Registers and Branch Address Calculator modules.

The General Purpose CPU Registers module, shown in Figure 7, consists of 32 32-bit registers, with register 0 always returning 0 and registers 1-31 being general purpose registers.

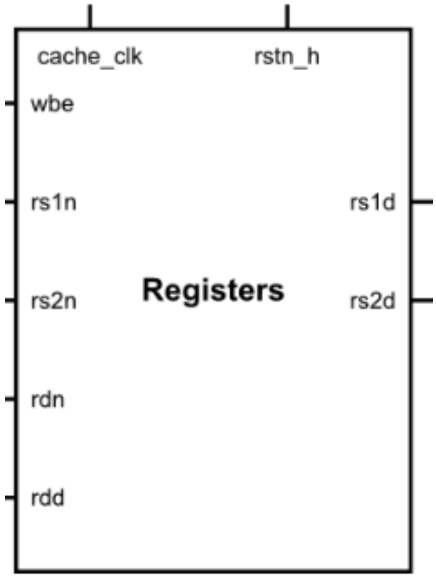

Figure 7: General Purpose CPU Registers Module

The General Purpose CPU Registers module takes in the register numbers for the first and second registers specified by the instruction in the ID stage and outputs the data values held in the registers. It also takes in the number and data to write for the destination register specified by the instruction in the WB stage and writes said data to the specified register. Writing to the destination register can be enabled or disabled using the wbe control signal.

The Branch Address Calculator module, shown in Figure 8, is used to calculate the next value for the program counter based on the branch predictor's prediction.

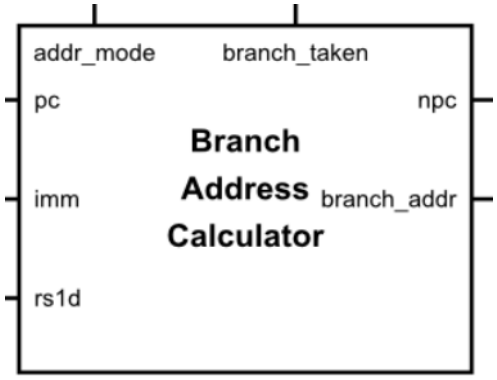

Figure 8: Branch Address Calculator Module

The addr_mode control signal can be either 0 or 1 and is used to specify how the address we will go to should a branch be predicted is calculated. The branch_taken input comes from the branch predictor and specifies whether or not to branch. Table 5 shows the two different methods of constructing the branch address based on the addr_mode control signal and Table 6 shows how the branch_taken input specifies the next program counter values where 0 means no branch and 1 means branch.

Table 5: addr_mode Control Signal Function

addr_mode	**Branch Address (branch_addr)**
0	pc + imm
1	imm + rs1d

Table 6: branch_taken Input Signal Function

branch_taken	**Next PC Value (npc)**
0	pc + 4
1	branch_addr

3.4. EXECUTION STAGE

The EX stage consists of the ALU and Branch Evaluator modules.

The ALU module, shown in Figure 9, is responsible for executing the various logic operations performed by the CPU. The logical operations supported by the ALU are as follows: Addition, subtraction, bitwise AND, bitwise OR, bitwise XOR, signed set on less than, unsigned set on less than, logical left shift, logical right shift, and arithmetic right shift.

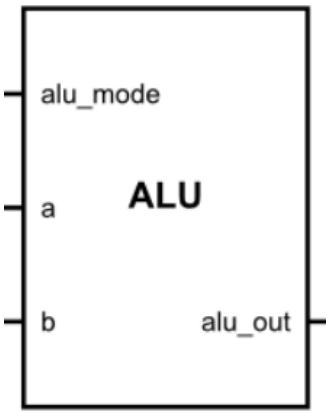

Figure 9: Arithmetic Logic Unit Module

The ALU takes in two operands, a and b, and performs the logical operation specified by the hex value of the 6-bit alu_mode control signal on them and then outputs the result. Table 7 shows the hex values corresponding to each operation.

Table 7: alu_mode Control Signal Function

alu_mode	**ALU Output (alu_out)**
0x00	a + b
0x20	a - b
0x07	a & b
0x06	a | b
0x04	a ^ b
0x02	signed a < signed b
0x03	a < b
0x01	a << b[4:0]
0x05	a >> b[4:0]
0x25	signed a >>> b[4:0]

Despite only having ten operations, 6-bit values are used for the ALU op-codes as it is easy to construct the specified values in Table 7 directly from the 32-bit instruction, so no extra conversion is needed.

The Branch Evaluator module, shown in Figure 10, is used to properly evaluate conditional branch instructions to check against the branch predictor's prediction.

Figure 10: Branch Evaluator Module

The branch_cond control signal ranges from 0 to 3 and specifies the current instruction's branch condition. For instructions with conditional branches, the ALU is used to evaluate the conditional so the ALU output is used to determine the correct action to take. Table 8 shows the branching types specified by the branch_cond control signal.

Table 5: branch_cond Control Signal Function

branch_cond	**act_taken**	**Corresponding Conditional**
0	0	Never Branches
1	|alu_out	Branch <, Branch !=
2	~|alu_out	Branch >, Branch ==
3	1	Always Branches

3.5. MEMORY STAGE

The MEM stage consists of the Data Cache module, shown in Figure 11, which is a connection module that contains the Data Cache Manager and L1 Data Cache modules.

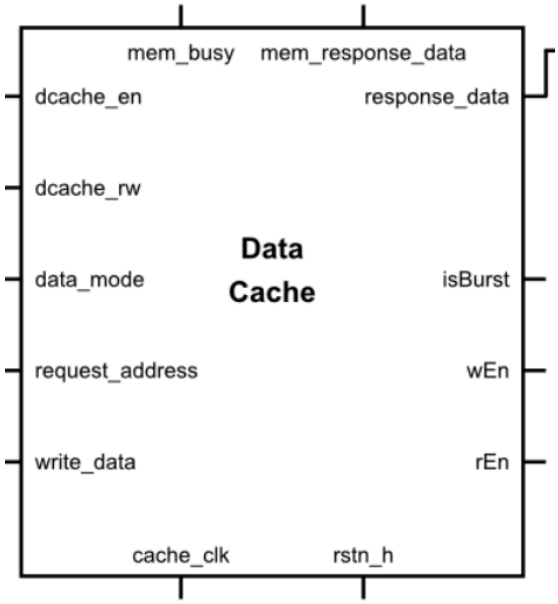

Figure 11: Data Cache Module

The Data Cache Manager module, much like the Instruction Cache Manager module, serves as a signal translation layer between the CPU, memory, and, in this case, the L1 Data Cache. The Data Cache Manager's primary function is to convert memory requests from the L1 Data Cache into the correct format for AMBA.

The L1 Data Cache is 4KiB in size and is 2-way associative, with a look-through read policy, write-back write policy, and a least recently used replacement policy. The block size of the cache is 4 bytes since our words are 32 bits, and with 2-way associativity, the cache contains 512 sets, each containing 2 32-bit blocks.

Notably, the data_mode input can be used to specify the number of bytes to write to a memory address. A data_mode value of 0 overwrites the least significant byte of the data at the specified memory address with the least significant byte of the input data, a data_mode value of 1 does this for the least

significant 2-bytes, and a data_mode value of 2 overwrites the entire 4-byte word.

Since the L1 Data Cache has a write-back write policy and not a write-through write policy, the L1 Data Cache handles all of the logic related to the data_mode input so that no extra logic needs to be added to the AMBA memory control. The L1 Data Cache only sends the resulting final 4-byte word of a write operation to memory, so from the memory's perspective all writes are the same size, a full word.

3.6. WRITE BACK STAGE

The WB stage consists of the General Purpose CPU Registers module, which it shares with the ID stage and as such has already been covered. The ID stage portion of the module is reading from the specified first and second registers while the WB stage portion is writing to the specified destination register.

3.7. BRANCH PREDICTION AND CONTROL

The modules that make up the system for branch prediction and branch control are the Pipeline Reset, Branch Predictor, and Branch Manager modules. It should be noted that the Branch Address Calculator and Branch Evaluator nodules are also a significant part of the branch prediction and control system but have definite pipeline stages they belong to, unlike the other branch modules.

The Pipeline Reset module, shown in Figure 12, is what provides the input signal for the Program Counter of what the next execution address is. It is also responsible for flushing the entire pipeline back and resetting it back to the proper next execution address following a branch prediction mistake.

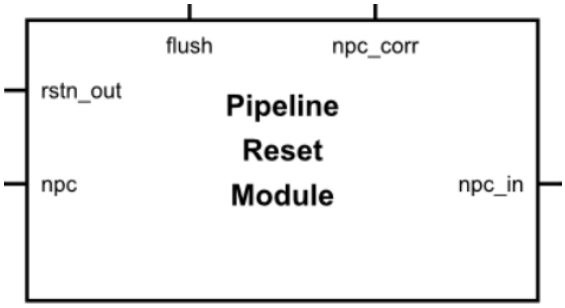

Figure 12: Pipeline Reset Module

The flush and npc_corr lines come from the Branch Manager module, with the flush line indicating that a pipeline flush needs to be executed and npc_corr being the correct next execution address for the mistaken prediction. If the flush line is set the Pipeline Reset module will pull the rstn line low, resetting the relevant modules, and then pass the npc_corr value to the program counter.

When the branch prediction result is correct, so there is no need to flush the pipeline, the Pipeline Reset module simply passes the next execution address received from the Branch Address Calculator module onto the Program Counter.

The Branch Predictor module, shown in Figure 13, handles the branch prediction logic for the CPU. It is a 2-bit predictor where the predicted branch outcome only changes after two incorrect predictions in a row.

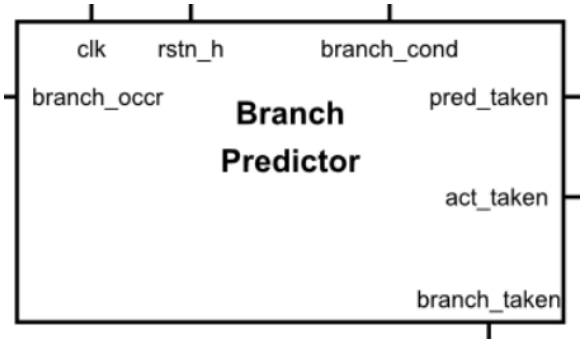

Figure 13: Branch Predictor Module

This module is a combination of combinations and sequential logic, with a 2-bit branch_occr control signal used to specify if a prediction is needed for the current instruction, where 0 means the instruction never branches, 1 means it always branches, and 2 means its a conditional branch and a prediction is needed. The way the branch_occr control signal is setup means that the most significant bit specifies whether or not a prediction is needed with the least significant bit specifying whether or not an instruction not needing a prediction does not branch or branches.

Combinational logic is used to set the branch_taken output according to the branch_occr control signal. If the most significant bit of branch_occr is not set, the branch_taken output is just set to the least significant

bit of branch_occr. If the most significant bit is set, the branch_taken output is set to the predicted result, which is determined by sequential logic.

Two 1-bit registers are used by the sequential logic system to make a branch prediction. One register stores the current prediction value, curr_pred, and the other stores whether or not the previous prediction was incorrect, incorrect_pred.

The pred_taken and act_taken inputs, along with the branch_cond control signal come from the EX stage and are used to determine whether or not the previous prediction was incorrect and if so whether or not the current prediction value needs to be changed. The pred_taken signal is the predicted branching result for the instruction that just moved out of the EX stage to the MEM stage while the act_taken input is the correct evaluated branching result for that same instruction. The branch_cond control signal ranges from 0 to 3 with 0 meaning the instruction never branches, 1 and 2 corresponding to different conditional branches, and 3 meaning the instruction always branches. The branch_cond control signal is used rather than the branch_occr control signal since the branch_occr control signal is set according to the instruction that just moved out of the ID stage into the EX stage which a prediction is currently being made for, while the branch_cond control signal is set according to the instruction that just moved out of the EX stage in the MEM stage which is the previous instruction evaluated by the branch predictor and so the one that has had its conditional properly evaluated and so can be checked.

For the Branch Predictor, a bitwise XOR is performed on the branch_cond control signal to determine if a prediction was made for the previous instruction. This operation will result in 0 for branch_cond values of 0 and 3, the cases where no prediction was made, and 1 for branch_cond values of 1 and 2, the cases where a prediction was made. If it is determined that no prediction was made for the previous instruction, the curr_pred and incorrect_pred register values are kept the same. If it is determined that a prediction was made for the previous instruction, the pred_taken and act_taken XORed together to check whether or not they are the same. If the resulting value is 0, the two inputs are the same meaning the previous prediction was correct; as such the curr_pred register is left unchanged and the incorrect_pred register is reset to 0. If the resulting value is 1, the two inputs are not the same meaning the previous prediction was incorrect and what happens next depends on the current value of the incorrect_pred register. If the incorrect_pred register is 0, it is set to 1 and the curr_pred register is unchanged. If the incorrect_pred register is 1, the curr_pred register is set to the opposite of its current value and the incorrect_pred register is unchanged.

An important note is that upon switching the prediction value the incorrect_pred register is not reset to zero since if the next prediction is also incorrect, that still counts as having two incorrect predictions in a row. The result is that for n incorrect predictions in a row, the prediction value is flipped n-1 times rather than n/2 times.

The Branch Manager module, shown in Figure 14, is responsible for alerting the Pipeline Reset module that the pipeline needs to be flushed and providing that module with the correct next execution address.

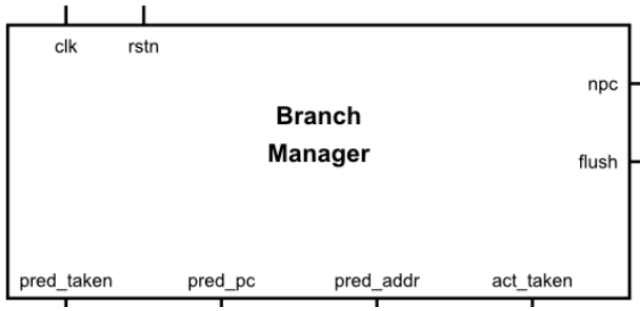

Figure 14: Branch Manager Module

Much like with the Branch Prediction module, the Branch Manager module uses the pred_taken and act_taken inputs to determine if an incorrect prediction was made. If the two inputs are not equal, meaning an incorrect prediction was made, and the internal restart register is not set, the flush output is set high and the npc output is set depending on the act_taken input. Table 6 shows the npc output values.

Table 6: act_taken Input Signal Function

act_taken	**npc**
0	pred_pc + 4
1	pred_addr

The pred_pc input is the program counter value for the instruction we are checking the prediction for and the pred_addr input is the branch address for that instruction. Act_taken gives the correct result of the conditional branch so determines whether we reset to the next sequential address or to the branch address.

If the pred_taken and act_taken inputs are equal, or the restart register is set, the flush output is set low and the npc output is set in the same way but using the pred_taken input instead of the act_taken one. The npc output here could be anything though as it only goes to the npc_corr input of the Pipeline Reset module which only reads that input if the flush line is set, which it isn't in this case. The reason for not just setting the npc output to some constant here is in case the value is ever needed by some module added in future iterations or by an improved version of the Pipeline Reset module.

The restart register is set high upon a reset of the module and then set low the next clock cycle, with it serving to prevent an accidental flush after a reset. After a reset both pred_taken and act_taken are designed to end up low, preventing a flush, but if the clock is pulsed too quickly after a reset the act_taken signal may not get set low in time since it is not set directly by a register but rather by the combinational logic of the ALU and Branch Evaluator modules. If the reset occurred when the act_taken signal was set high this could pose a problem. Since the pipeline will never actually need to be flushed on the first clock cycle after a reset anyway, adding the restart register has no real downside and so was determined to be worth it to prevent this potential problem.

3.8. HAZARD DETECTION

When the instruction in the IF stage poses a read-after-write or write-after-write hazard for the CPU registers, we need to stall that instruction until the instruction it depends on exits the pipeline. A similar rule is needed for branching hazards since we could potentially load an errant instruction into the IF stage. As such, combinational logic in the General Control module is used to check if the instruction in the IF stage is a branching instruction or otherwise dependent on any instruction currently in the other stages of the pipeline. If there is a branching instruction or dependency, the General Control module's internal hazard register is set and a NOP is inserted into the pipeline to stall execution; otherwise, execution continues as normal with the hazard register not set.

The hazard detection logic is implemented combinationally and not sequentially since the hazardous instruction in the IF stage needs to be stalled in that stage for as long as the hazard is present. Since instructions advance along the pipeline at the rising edge of the clock, the logic to stall the instruction must execute before the next clock pulse.

The hazard detection system does not stall for write-after-read hazards since the design has no instruction reordering meaning there is no way for a subsequent instruction to read data from the registers before the proceeding instruction can write the data. The system also does not consider memory hazards since only the instruction in the MEM stage can access memory, and without reordering, the initial instruction will always finish working with memory before the dependent instruction can access memory.

It should be noted that a read-after-write memory hazard could occur if the area of memory storing the current program is written to. An instruction could modify the value at a sufficiently close execution address such that the instruction at that address is fetched from memory before it can be overwritten. However, since we are not currently supporting self-modifying code, detecting read-after-write memory hazards is unnecessary.

Table 7 shows the different instruction types and the potential hazards they pose when in the IF stage. The data hazards only concern the CPU registers and not memory for the reasons mentioned above.

Table 7: Potential Hazards Per Instruction Type at IF Stage

Instruction Type	Read After Write	Write After Write	Branch Hazard
R	Yes	Yes	No
I Type 1	Yes	Yes	No
I Type 2	Yes	Yes	No
I Type 3	Yes	Yes	Yes
S	Yes	No	No
B	Yes	No	Yes
U	No	Yes	No
J	No	Yes	Yes
NOP	No	No	No

Table 8 shows the potential hazards the various instruction types pose when later in the pipeline.

Table 8: Potential Hazards Per Instruction Type at Later Stages

Instruction Type	Read After Write	Write After Write
R	Yes	Yes
I Type 1	Yes	Yes
I Type 2	Yes	Yes
I Type 3	Yes	Yes
S	No	No
B	No	No
U	Yes	Yes
J	Yes	Yes
NOP	No	No

The hazard control system uses the logic shown in Figure 15.

- If (`ins[6:0]` == I3, B, J) and (`hazard` == 0)
  - `hazard` = 1
- Else if `ins[6:0]` == R, I1, I2, I3, S, B, U, J
  - If (`ID_ins[6:0]` == R, I1, I2, I3, U, J) and ((`ID_ins[11:7]` == `ins[11:7]`) or (`ID_ins[11:7]` == `ins[24:20]`) or (`ID_ins[11:7]` == `ins[19:15]`))and (`ID_ins[11:7]` != 0)
    - `hazard` = 1
  - If (`EX_ins[6:0]` == R, I1, I2, I3, U, J) and ((`EX_ins[11:7]` == `ins[11:7]`) or (`EX_ins[11:7]` == `ins[24:20]`) or (`EX_ins[11:7]` == `ins[19:15]`))and (`EX_ins[11:7]` != 0)
    - `hazard` = 1
  - If (`MEM_ins[6:0]` == R, I1, I2, I3, U, J) and ((`MEM_ins[11:7]` == `ins[11:7]`) or (`MEM_ins[11:7]` == `ins[24:20]`) or (`MEM_ins[11:7]` == `ins[19:15]`))and (`MEM_ins[11:7]` != 0)
    - `hazard` = 1
  - If (`WB_ins[6:0]` == R, I1, I2, I3, U, J) and ((`WB_ins[11:7]` == `ins[11:7]`) or (`WB_ins[11:7]` == `ins[24:20]`) or (`WB_ins[11:7]` == `ins[19:15]`))and (`WB_ins[11:7]` != 0)
    - `hazard` = 1
- Else
  - `hazard` = 0

Figure 15: Pseudo-Code for Hazard Detection Logic

As shown in Figure 15, the hazard control logic first looks for a potential branch hazard, and so checks whether or not the instruction in the IF stage is an I type 3, B, or J instruction. Provided the hazard bit is not set, meaning this instruction has just been loaded into the IF stage, the hazard bit is set and a nop is inserted into the pipeline. The hazard bit is checked to ensure that the branch instruction is only stalled for a single clock cycle, and not forever.

If no branch hazard is found, the hazard control logic then moves to check if the instruction in the IF stage poses a potential data hazard, meaning it checks for R, I type 1, I type 2, I type 3, S, B, U, and J type instructions. When such an instruction is found, each stage is checked for any R, I type 1, I type 2, I type 3, U, or J type instructions whose destination register matches any of the accessed registers in the IF stage instruction. If such a match is found, the hazard bit is set and a nop is inserted, ensuring that the pipeline is stalled until the offending instruction exists the pipeline. If no hazards are found then the hazard bit is set to zero and pipeline execution resumes as normal.

3.9. CONTROL SIGNALS

The General Control module, shown in Figure 16, is also responsible for decoding instructions and sending out the various module control signals explained in the above sections for the instruction at each stage at the right times.

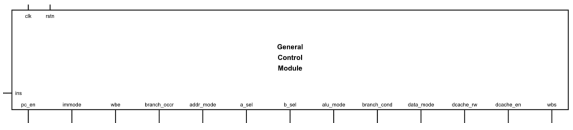

Figure 16: General Control Module

Tables 9 through 13 show the control signals for the IF, ID, EX, MEM, and WB stages respectively, and how they are set for the various instruction types.

Table 9: IF Control Signals

Instruction Type	**immode**
R	0
1 Type 1	1
I Type 2	1
I Type 3	1
S	2
B	3
U	4
J	5
NOP	0

Table 10: ID Control Signals

Instruction Type	**addr_ mode**	**branch _occr**	**a_sel**	**b_sel**
R	0	0	0	0
1 Type 1	0	0	0	1
I Type 2	0	0	0	0
I Type 3	1	1	1	2
S	1	0	0	0
B	0	2	0	0
U	0	0	ID_ins[5:4]	3
J	0	1	1	2
NOP	0	0	0	0

Table 11: EX Control Signals

Instruction Type	**alu_mode**	**branch_comd**
R	EX_ins[31:25] + EX_ins[14:12]	0
1 Type 1	If EX_ins[14:12] == 0x5, alu_mode =EX_ins[31:25] + EX_ins[14:12], otherwise alu_mode = EX_ins[14:12]	0
I Type 2	0	0
I Type 3	0	3
S	0	0
B	If EX_ins[14:12] == 4 or 5 alu_mode = 0x02, if EX_ins[14:12] == 6 or 7 alu_mode = 0x03, otherwise alu_mode = 0x20	If EX_ins[14:12] == 1 or 4 or 6 branch_cond = 1, else brach_cond = 2
U	0	0
J	0	3
NOP	0	0

Table 12: MEM Control Signals

Instruction Type	data_mode	dcache_ rw	dcache_ en
R	0	0	0
1 Type 1	0	0	0
I Type 2	0	0	1
I Type 3	0	0	0
S	MEM_ins[14:12]	1	1
B	0	0	0
U	0	0	0
J	0	0	0
NOP	0	0	0

Table 13: WB Control Signals

Instruction Type	wbs	wbe
R	3	1
1 Type 1	3	1
I Type 2	WB_ins[14:12]	1
I Type 3	3	1
S	0	0
B	0	0
U	3	1
J	3	1
NOP	0	0

3.10. CLOCKS AND RESET LINES

Two different clock lines are used in the core design, clk and cache_clk. The cache_clk line is used for the CPU registers and caches while the clk line is used for everything else. The two clock lines are needed since the registers and caches need to wait for the latches to update before they themselves can update. As such, the cache_clk is inverted from the clk line so that in-between each rising edge of the clk line is a rising edge of the cache_clk line.

The core also has two different but related reset lines, the rstn line and the rstn_h line, both of which are active low. The rstn_h line is the master reset line and whenever it is pulled low the rstn line will also be pulled low; however, pulling the rstn line low will not pull the rstn_h line low. This reset line system is needed since the pipeline is flushed via the rstn line but the registers in the Branch Predictor, Instruction Cache, Data Cache, and General Purpose CPU Registers modules need to maintain their values after a pipeline flush to properly execute the correct post-branch instruction. These modules, however, still need to have a way to be reset so they use the master rstn_h line which is used when one wants to reset the entire state of the core such as during initialization.

3.11. CONNECTION MODULES

Six higher-level connection modules were designed to simplify the process of connecting and testing related modules. These modules are the IF connection module, ID connection module, EX connection module, MEM connection module, Branching Control connection module, and the Top Level module. Figure 17 shows the connection module each module belongs to, with red being IF, orange being ID, yellow being EX, green being MEM, and purple being Branch Control. The General Control module and MEM/WB latch are given their own colors to signify they belong to no connection module and are instead directly connected in the Top Level module alongside the connection modules.

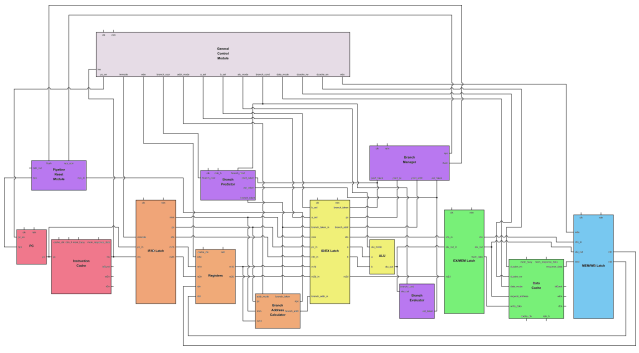

Figure 17: CPU Block Diagram Connection Modules

4. IMPLEMENTATION WORK

A given component's completion level is broken down into 5 categories. The first category is “needs to be outlined” which means a component exists in name only and has not yet been fully designed. The second category is “needs to be implemented” which indicates that a component has been designed and outlined but has not yet been implemented in SystemVerilog nor had its functionality tested. The third category, “needs to be connected”, is for components that have been implemented and tested on their own, but belong to a higher-level module that has yet to be implemented and tested itself. The fourth category is known as “connected” and indicates that the higher-level module containing the component has been implemented and tested, meaning the individual component has had its functionality verified when in concert with other related components. Modules in the “connected” category still need to be tested as a part of the complete CPU, however. Finally, the fifth category “complete” is for modules that have been tested as a part of the complete CPU and determined to operate correctly.

Figure 18 shows the Core Design and Implementation GitHub project which is used to track the progress of core related components.

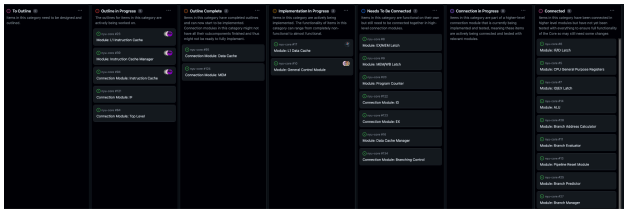

Figure 18: Core Design and Implementation GitHub project

Figure 18 breaks the categories down further to indicate whether or not a given component is actively being worked on.

Currently, the components that are left to outline are the L1 Instruction Cache module, Instruction Cache Manager module, Instruction Cache connection module, IF connection module, and the Top Level module.

The components that are outlined but not yet implemented and tested are the L1 Data Cache module, the General Control module, the Data Cache connection module, and the MEM connection module.

The components that have been implemented and tested but are left to be connected within higher-level connection modules are the EX/MEM Latch module, the MEM/WB Latch module, the Program Counter module, the Data Cache Manager module, the ID connection module, the EX connection module, and the Branching Control connection module.

Finally, the components that have been connected but are waiting for the entire CPU to be implemented to ensure their full functionality are the IF/ID Latch module, the CPU General Purpose Registers module, the ID/EX Latch module, the ALU module, the Branch Address Calculator module, the Branch Evaluator module, the Pipeline Reset module, the Branch Predictor module, and the Branch Manager module.

5. CONCLUSIONS

The initial goal when the NYU Processor Design Team started was to have a functional taped-out SoC in our hands within three semesters, meaning by the end of Spring 2024 since the team started in Spring 2023. Needless to say, that goal was not met, with significant work left to be done on the CPU core as well as on other SoC components such as AMBA.

Despite that, significant progress was made on the core. The design is complete apart from the instruction cache, and most of the modules have been implemented and tested.

6. RECOMMENDATIONS

Once all of the implementation work is completed, significant testing needs to be done before tapeout. Ideally, this includes synthesizing and testing the full design on an FPGA, but given the idiosyncrasies of timing on FPGAs as well as the size of the FPGA that would be needed to simulate the full SoC, a more realistic option would be to test individual

subcomponents of the SoC on FPGAs rather than the entire thing.

There are also further optimizations that can be done for various components of the CPU core, particularly the branch prediction and hazard detection systems. Various branch prediction methods should be tested to determine the optimal one given the design of the CPU and the types of programs expected to be run on it. In terms of the hazard detection logic, the system currently stalls for both read-after-write and write-after-write data hazards, but given that the only stage in which an instruction writes to a register is the WB stage, it may be entirely unnecessary to stall for write-after-write data hazards. This is one of many potential optimizations that could be made to the hazard detection system but that requires thorough testing to ensure it does not cause any issues.

7. REFERENCES

[1] Edited by Andrew Waterman et al., *Volume I: Unprivileged ISA Document Version 20191213*, University of California, Berkeley, 13 Dec. 2019, riscv.org/wp-content/uploads/2019/12/riscv-spec-20191213.pdf

8. APPENDIX

The SystemVerilog modules, C++ test code, and documentation for the core design can be found at the NYU Processor Design Team nyu-core GitHub repository linked here: https://github.com/NYU-Processor-Design/nyu-core