Unemployed Trio

Phillip Xie (Our proud leader)	px40
Jang Uk (Davd) Park	jup2
Brian (HyukJoo) Kim	brianhk2

Introduction

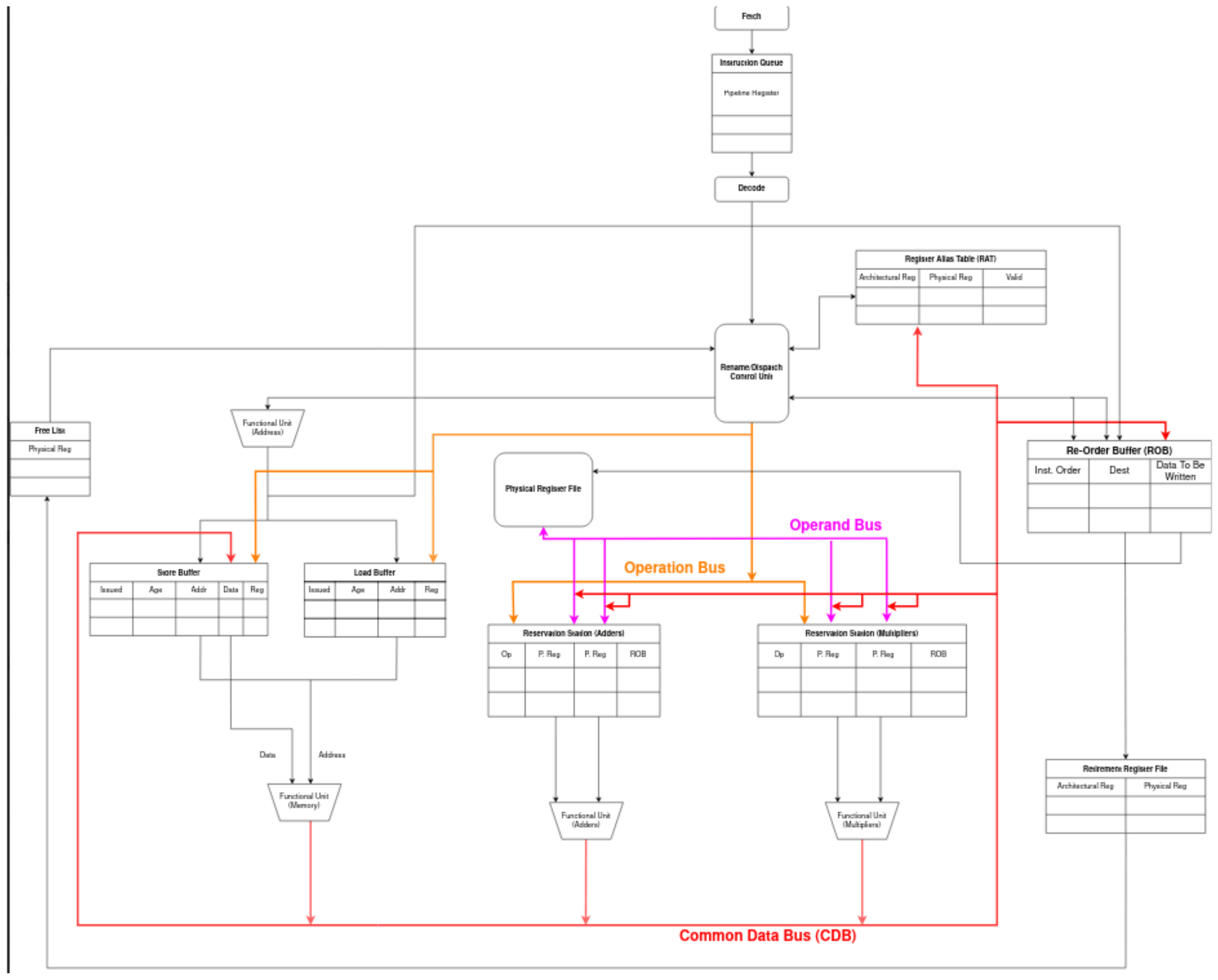
The main technical thrust of our project revolves around the design and implementation of a high-performance Out-of-Order (OOO) RISC-V Processor, incorporating advanced features such as Explicit Register Renaming, GShare Branch Predictor, Branch Perceptron, Pipelined I-Cache, Wallace Multiplier, and DADDA Multiplier. Our endeavor delves into the realm of computer architecture, aiming to construct a processor capable of executing instructions out of their original sequential order, thereby maximizing performance through instruction-level parallelism. This report encapsulates our journey through the design, implementation, and analysis of this processor, highlighting key milestones, advanced features, design tradeoffs, and performance evaluations.

Project Overview

Our project is focused on the construction of an Out-of-Order RISC-V Processor with Explicit Register Renaming. Our endeavor was motivated by the goal of maximizing performance within a specified timeframe. To achieve this, we organized our efforts into distinct areas, including instruction processing/dispatching, OOO implementation, and instruction commitment. Collaborative debugging sessions, conducted in pairs, proved instrumental in swiftly identifying and resolving issues, enhancing both productivity and problem-solving capabilities. Noteworthy among our achievements is the reduction of the clock cycle to 1.33 ns, demonstrating significant performance optimization. Our report provides comprehensive insights into our methodologies, challenges encountered, and the innovative solutions devised throughout this project.

Checkpoint 1

Checkpoint 1 marks a significant milestone in the evolution of our processor design, with a focus on the Fetch Stage and the Instruction Queue. At this stage, our work is primarily divided into these two components. The Instruction Fetch component has been implemented with base functionality, initializing the Program Counter (PC) to address 60000000 upon reset and incrementing it by 4 on each cycle. An important aspect is the handling of the full signal sent by the queue, ensuring proper synchronization. The behavior observed involves latching the fetched instruction and stalling the PC value as necessary. The Instruction Queue operates as its name suggests, efficiently managing the enqueueing and dequeueing of instructions. To verify complete functionality, we have developed a separate test bench focusing on various aspects, including correct instruction fetch, proper latching of instructions, queue behavior concerning both enqueueing and dequeueing, and ensuring proper generation of full and empty signals. This comprehensive testing approach allows us to validate the functionality of each component and ensure adherence to specifications. As we transition to subsequent checkpoints, we anticipate further refinement and enhancement of our processor design, building upon the solid foundation established in this initial milestone.



Checkpoint 2

After our Checkpoint 1 milestone had come to a close, it was time for our group to build the rest of the processor. Our group worked to build each module starting from the Rename/Dispatch all the way to our free list. A more in-depth description of each module is written below:

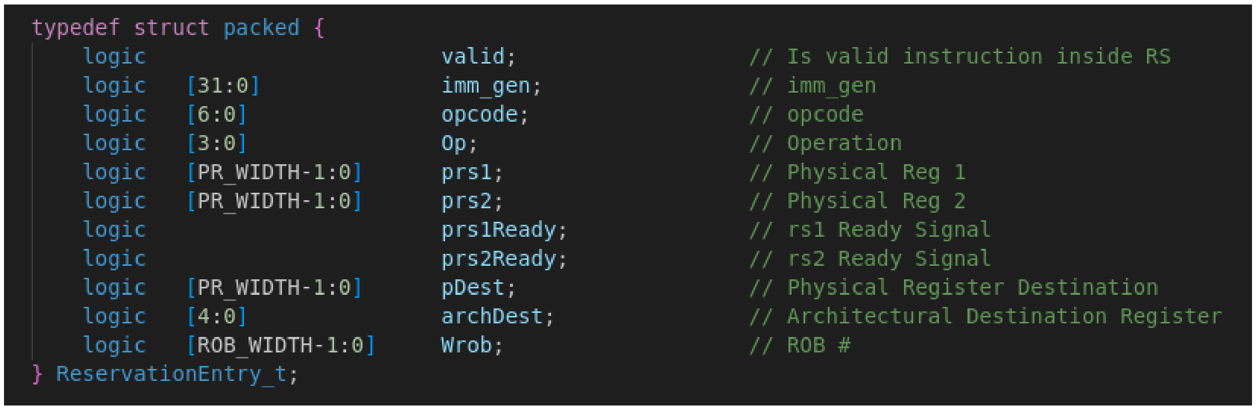
Decode:
In our decode stage module, we focus on efficiently decoding incoming instructions. We break down each instruction into essential signals and fields required for further processing in the pipeline. We ensure precise extraction of opcode, registers, immediate values, and control signals following RISC-V specifications. Additionally, we manage the instruction queue, ensuring instructions are dequeued when the empty signal is not held high. We also handle potential stall conditions, indicated by signals like re_dis_stall, to maintain pipeline efficiency.

Rename/Dispatch:

In our Rename/Dispatch stage, we play a crucial role in managing the flow of instructions through the pipeline and handling various conditions that may impact execution. We generate the re_dis_stall signal, triggered when reservation stations or the Reorder Buffer (ROB) are full, or when the free list is empty. During renaming, we allocate physical registers from the free list and associate them with architectural destination registers. Additionally, we fetch physical registers from the Register Alias Table (RAT) corresponding to architectural operands. For dispatching, we set the appropriate write/enable bits for the ROB and corresponding reservation stations, as well as configure reservation station and ROB entries. We update the RAT to reflect the new physical register mappings. In Check 3 updates, for branch instructions, we forward the calculated new address to the ROB and reset the destination register. For Jal instructions, treated as add instructions, we set the immediate value as the current PC + 4, with operands set to 0, passing the calculated address to the ROB. Similarly, Jalr instructions are treated as add instructions, with the immediate value set to the previous PC + 4, and the register value saved for later use in the ROB to compute the branch address.

Reservation Station (Add/Multiply):

Both of these Reservation stations were designed to hold their respective instructions (Multiply Reservation Station for Multiply Instructions and Add Reservation Station for the ALU instructions). Due to the similarity in their design, I decided to instantiate both inside the same module with similar actions based on the signals that were given. For instance, on reset, both stations were set to 0's. When either an Add or Multiply Instruction was passed in from the Rename/Dispatch Module, the corresponding index of the Reservation was populated with information regarding the Physical Register Destination, Rs1 addr, Rs2 addr, information on whether or not the registers are ready with data, and so on. A screenshot of how each index was populated is included below:


```
typedef struct packed {
    logic                       valid;              // Is valid instruction inside RS
    logic   [31:0]              imm_gen;            // imm_gen
    logic   [6:0]               opcode;             // opcode
    logic   [3:0]               Op;                 // Operation
    logic   [PR_WIDTH-1:0]      prs1;               // Physical Reg 1
    logic   [PR_WIDTH-1:0]      prs2;               // Physical Reg 2
    logic                       prs1Ready;          // rs1 Ready Signal
    logic                       prs2Ready;          // rs2 Ready Signal
    logic   [PR_WIDTH-1:0]      pDest;              // Physical Register Destination
    logic   [4:0]               archDest;           // Architectural Destination Register
    logic   [ROB_WIDTH-1:0]     Wrob;               // ROB #
} ReservationEntry_t;
```

Our reservation station was populated randomly using a for-loop. Whenever an open index was found, the new entry that was passed from the Rename/Dispatch module would be populated at that specific index. Another for-loop was used to determine if an entry was ready to leave the station and move onto the ALU unit/Multiplier unit. The state of being “Ready” was determined by the psr1ready and psr2ready signals which are populated by either the Rename/Dispatch module if the physical registers are valid or populated by the CDB if the dependent physical register is currently being calculated by one of the functional units (Memory, ALU, Multiplier). Although it can be assumed from the previous information, the psr1Ready and psr2Ready signal are set using a for loop with each and every CDB value being matched with each index of the reservation station to see if there is a match in physical register.

Tests: Custom Tests were made to ensure that each entry inside the Reservation Station was dequeued at some points and the psr1Ready/psr2Ready signals are set with dependent CDB values.

ROB:

The ROB was designed as a queue that dequeues once the output for that particular instruction is ready and the ROB head pointer is also pointing at that instruction. Once that instruction is ready to be committed, it is committed combinational-ly to RVFI and dequeued in the following clock cycle. Once dequeued, the head pointer is incremented once and waits for the next instruction on the queue to receive its output from the CDB. The CDB’s functionality is very similar to that of the CDB’s functionality inside the reservation stations. There is a combinational for loop that iterates across all ROB entries and populates the ROB valid signal once there is a valid CDB data that has matching ROB index numbers with that of the ROB index.

Tests: Custom Tests were made to ensure that each entry inside the ROB was dequeued once the head pointer is pointing to an instruction that is ready to be committed. I also made tests to make sure each ready entry is being committed combinational-ly to RVFI and branch bits were being sent out when they were supposed to.

RRF:

The RRF was designed to keep track of the physical register/architectural register mappings on commit. This information is necessary because on a branch instruction that is taken, we needed to clear all the structures within our processor (ROB, Reservation Stations, instruction queue) and we also had to copy paste our RRF into the RAT to push the mappings that were committed to be the current mappings.

Tests: Created Test cases to see that committed instructions and its physical registers were correctly mapping to its corresponding index in the RRF. The visual commits can be seen through Verdi waveform as well.

RAT:

The RAT was designed to keep track of the physical register/architectural register mappings on instruction dispatch to the Reservation Station. This information is necessary because consecutive instructions need to know which registers have valid mappings, so that physical registers are not used for a particular instruction when they are not supposed to. The RAT provides Rename/Dispatch module with the correct physical registers used for an instruction.

Tests: Created Test cases to see that valid/invalid bits for a particular register were set once an instruction was dispatched to the Reservation Station.

Free List:

This module manages registers that are available for use. It employs two pointers, head_ptr and tail_ptr, to monitor the boundaries of the free registers. Specifically, these pointers help track which registers are currently unoccupied and can be allocated for new operations.

For operations involving branching, a separate pointer, branch_head_ptr, is utilized. This pointer only increments upon the commit of a branch. It's important to note that there is no corresponding branch_tail_ptr because its behavior and characteristics mirror those of the tail_ptr. Both pointers—tail_ptr and the hypothetical branch_tail_ptr—are updated exclusively during a commit phase, aligning their functionality.

When a branch operation is initiated, the system adjusts the head_ptr to align with the branch_head_ptr. This adjustment is crucial as it allows the system to revert to a known good state without altering the sequence of committed and pending operations, thereby maintaining program correctness after a branch.

In contrast, jumping operations, such as those triggered by jal or jalr, behave slightly differently. These are akin to branching but include a commit followed immediately by a flush, encapsulated by the term "commit + flush." During such operations, the head_ptr is set to match the branch_head_ptr. This alignment ensures that the register state reflects the jump's effect, but the head_ptr itself does not change during a commit; this is identical to the behavior during a branch operation. However, the tail_ptr undergoes a specific change—it increments by one. This increment accounts for the register that is committed during the jump operation. Previously, there were issues with this approach, where the tail_ptr increment led to a situation where it erroneously skipped a slot. This caused an input of zero into the system, leading to errors. On closer examination, it was determined that the underlying cause of the issue was the absence of a commit operation during the jump. By ensuring that a commit takes place before the jump, and correctly adjusting the tail_ptr, these errors were rectified, ensuring that the module functions correctly without introducing gaps in the register allocation.

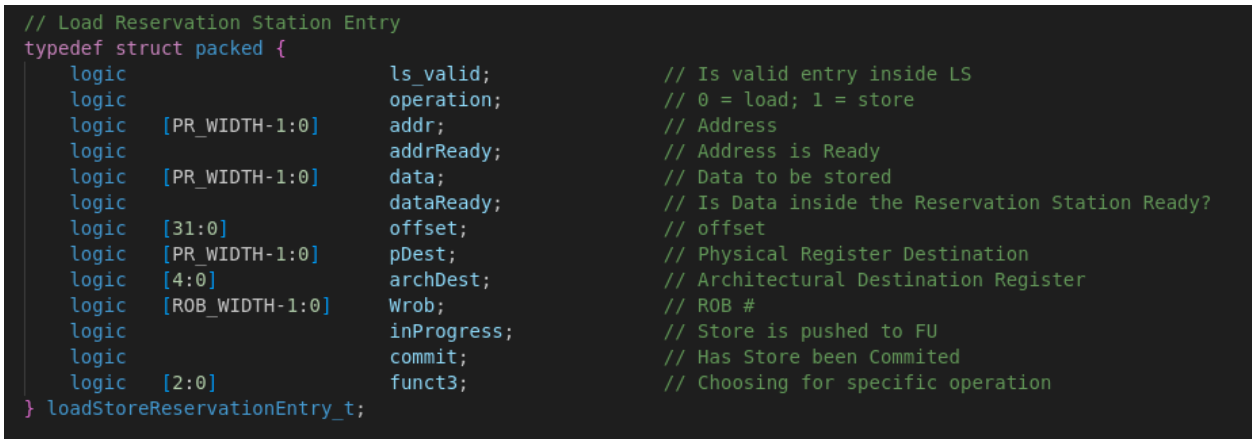
Physical Register File:

The phys_reg module is an essential component in microprocessors implementing Explicit Register Renaming. This module operates as a dynamic storage for actual data values mapped by the Register Alias Table (RAT), essentially functioning as a circular queue to manage read and write operations requested by reservation stations and branching units. It uses a variety of inputs including clock, reset, write enable, and several others to determine the physical registers that need to be read from or written to. Specifically, it resets all register values to zero on a reset command, writes new data to specified registers based on control signals that dictate write conditions, and simultaneously outputs the data stored in multiple physical registers to various execution units like ALU and memory access. The phys_reg module not only maintains the integrity of data within the processor's execution path but also ensures that the correct data is available at the right time for every part of the processor.

Checkpoint 3

Reservation Station (Load/Stores):

In our design, we decided to keep both Loads and stores inside the same Reservation Station. This way, we were able to ensure that Loads were not dequeued when there is a store that is being processed in the ROB/Memory functional unit. As for the overall design of the Load/Store Reservation Station, we made it into a queue so that Memory dependent instructions ensured correct calculations using correct memory values. Each Reservation Station entry was populated with information regarding its physical register destination, Address that will be accessed for reading/writing, as well as information regarding whether or not it has been pushed to the ROB or not. A more detailed view into what each entry holds can be seen below:



```
// Load Reservation Station Entry
typedef struct packed {
    logic                       ls_valid;             // Is valid entry inside LS
    logic                       operation;            // 0 = load; 1 = store
    logic   [PR_WIDTH-1:0]      addr;                 // Address
    logic                       addrReady;            // Address is Ready
    logic   [PR_WIDTH-1:0]      data;                 // Data to be stored
    logic                       dataReady;            // Is Data inside the Reservation Station Ready?
    logic   [31:0]              offset;               // offset
    logic   [PR_WIDTH-1:0]      pDest;                // Physical Register Destination
    logic   [4:0]               archDest;             // Architectural Destination Register
    logic   [ROB_WIDTH-1:0]     Wrob;                 // ROB #
    logic                       inProgress;           // Store is pushed to FU
    logic                       commit;               // Has Store been Commited
    logic   [2:0]               funct3;               // Choosing for specific operation
} loadStoreReservationEntry_t;
```

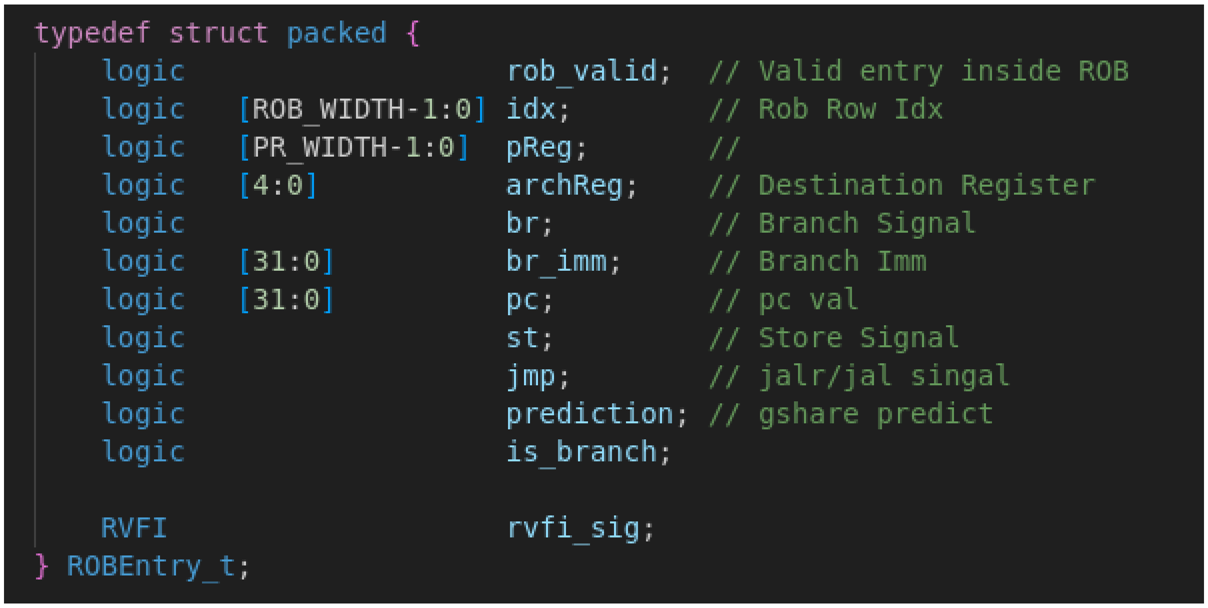
We implemented a special case for when stores are at the top of the queue. We decided to keep the store at the top of the queue even after its data has been passed onto the memory functional unit. This way, we prevent any other loads/stores from populating the ROB until this store commits. Once the store commits, I go to the Load/Store queue and check to see that the store that is about to commit is the same store that is on top of the Load/Store queue. If they are the same instruction, both are dequeued from their corresponding data structures.

Tests: Custom Tests were made to ensure that each entry inside the Reservation Station was dequeued at some points and the psr1Ready/psr2Ready signals are set with dependent CDB values. We also had to make sure the stores which are at the top of the queues did not deque when they are not supposed to.

ROB (Updated for Checkpoint 3):

The ROB has unique functionality when it comes to branches as well. Once the CDB value for a specific branch instruction makes it to the ROB, the expected outcome of the branch is determined by a specific bit. So, if that particular bit is high, this means that this branch is taken; else, not taken. The ROB entry also contains a specific entry that determines whether or not this branch was predicted to be taken/not taken. If the prediction matches up with the final outcome of the branch, nothing is done. However, if there was a missed branch (prediction val is low while branch outcome is high), the branch bit is set to high and the branch pc that is combinational-ly sent back to our fetch stage is that of the branch immediate value which was calculated in our Decode stage. If a false prediction happened, then this branch bit is set high again, but the branch pc is set to the branch isntruction's pc + 4. This is then the value of the pc value that should have been taken after the branch instruction.

In addition to the prediction bit and the branch outcome bit, the ROB entry has other elements including the ROB valid bit which tells the ROB head pointer that a particular instruction is ready to be committed. There are also other elements such as the physical register value, architectural register value, ROB idx, and others. The screenshot below gives a more in-depth view of which elements our ROB entries held.



```
typedef struct packed {
    logic                    rob_valid;  // Valid entry inside ROB
    logic   [ROB_WIDTH-1:0] idx;        // Rob Row Idx
    logic   [PR_WIDTH-1:0]  pReg;       //
    logic   [4:0]           archReg;    // Destination Register
    logic                   br;         // Branch Signal
    logic   [31:0]          br_imm;     // Branch Imm
    logic   [31:0]          pc;         // pc val
    logic                   st;         // Store Signal
    logic                   jmp;        // jalr/jal singal
    logic                   prediction; // gshare predict
    logic                   is_branch;

    RVFI                    rvfi_sig;
} ROBEntry_t;
```

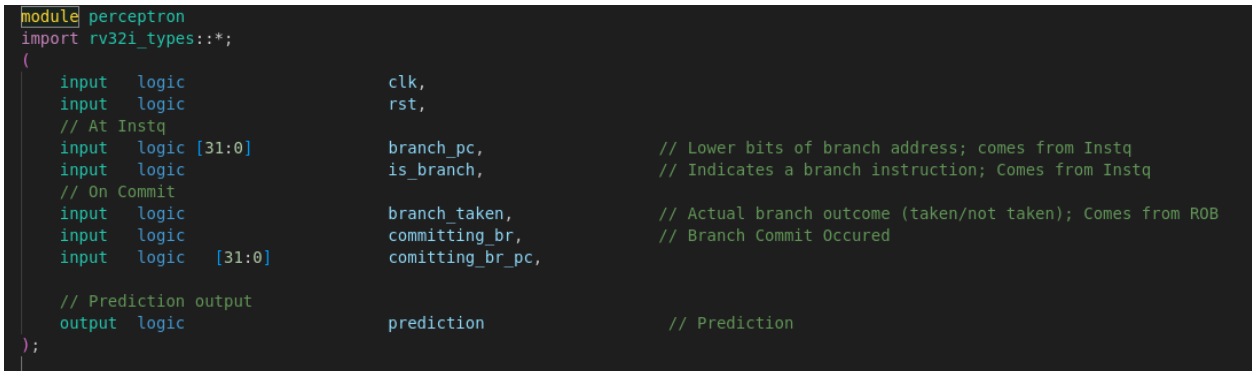
Cache (w/ cache arbiter)

This cache configuration comprises 16 sets, 4 ways, and 256 cache lines, and it is designed to interact with both the CPU and the cache_arbiter. Note that this description is based on the checkpoint 3 cache, not the one with the advanced features. Two instances of this cache, namely the inst_cache and the data_cache, are instantiated for handling different types of data. The cache's operation is segmented into five distinct states: idle, compare_tag, write_back, allocate, and stall. In the idle state, the cache awaits requests from the CPU. The compare_tag state involves checking if the requested data is already in the cache. If the cache contains modified data that needs to be written back to the bmem, it enters the write_back state. The allocate state is activated when new data needs to be saved from the bmem into the cache.

Finally, the stall state is required to ensure a one-clock delay to complete all data transfers within the cache, necessary due to the allocation process. Additionally, because of the characteristics of the bmem as described in the cache_arbiter documentation, data transfer between the bmem and the cache requires looping four times as the data is handled in 64-bit segments. This necessitates repeatedly sending request signals until the arbiter grants the cache permission to communicate with the bmem, ensuring the efficient management of data flow and cache operations. If there is a branch, the cache continues its work if its on allocate or write_back stage but for allocate state, it won't write to the cache nor will it become hit on the cache when it goes back to compare_tag state.

There is also a cache_arbiter module that is associated with the cache module. The cache_arbiter is designed to manage requests from both the imem_cache and the dmem_cache. It employs a round-robin algorithm to ensure that both the instruction cache inst_cache and the data_cache have equal opportunities to access the bmem. The bmem has specific characteristics that must be adhered to; notably, the input signal should only be active for one cycle when the bmem_ready signal is high. Additionally, the bmem incorporates a queue system that allows for multiple read operations to be returned in an out-of-order sequence, while multiple write operations are executed in an in-order fashion. However, we did not utilize that characteristic for this implementation of the CPU.

Branch Perceptron



```
module perceptron
import rv32i_types::*;
(
    input   logic                       clk,
    input   logic                       rst,
    // At Instq
    input   logic [31:0]                branch_pc,                  // Lower bits of branch address; comes from Instq
    input   logic                       is_branch,                  // Indicates a branch instruction; Comes from Instq
    // On Commit
    input   logic                       branch_taken,               // Actual branch outcome (taken/not taken); Comes from ROB
    input   logic                       committing_br,              // Branch Commit Occured
    input   logic   [31:0]              comitting_br_pc,

    // Prediction output
    output  logic                       prediction                   // Prediction
);
```

One advanced feature that my group decided to implement was the Branch Perceptron. I designed my branch predictor to hold 256 perceptrons with each perceptron holding 8 weights. Each weight has a width of 8 bits and a generic threshold was set to 8 bits. The threshold is maintained to determine whether the branch should be taken or not. A Global History Register also exists to keep track of the outcomes of the 8 most recent branches. The high-level overview of how my perceptron functions is as follows:

Advanced Design

1. The bottom bits of the branch program counter (branch_pc[9:2]) were used to hash into my Perceptron Table. Using bits 9:2 instead of 7:0 was a design choice that I made because I noticed some indices were not being used due to the pc being multiples of 4.
2. After fetching the weights from the specified index, I calculated $y = w0 + \sum xiwi$ with "w" representing the weights and "x" representing the lower 8 bits of the global branch history shift register.
3. If the value of y exceeds the given threshold of "x000F," I will predict the branch as being taken and relay the predicted branched pc to our "Fetch stage."
4. When the corresponding branch instruction is about to get committed, I will once again hash into the Perceptron Table to see the original prediction for the given branch. If the prediction and the actual result differ, I will update each weight in the hashed index by incrementing each update by 1 if the GHR index agrees with the Branch outcome. Else, decrement the weight.
5. If a mispredict was determined on commit, I would relay back the branch pc + 3'b100 (4) to the fetch stage and combinational-ly set that as the next pc to make up for the branch mispredict by my Perceptron.

Design Tradeoffs:

Although the Branch Perceptron has been very accurate as described in the table below with IPC values also showing slight improvement compared to our base design with no predictor, our clock period had to increase a lot (20k) due to the dot product calculation that was necessary for predicting a branch. In addition, the base processor + base cache had an area of ~150k, but the perceptron brought up the area by double (~300k). This brought down the performance of our processor, so we decided to not add the Perceptron into the final design. The increase in IPC was simply not beneficial when our Clock Period saw an increase of 1000% and area by 100%.

Performance Analysis:

Perceptron Accuracy/speed up on varying test cases vs No Predictor:

Test File	Total Branch	IPC	TP	FP	TN	FN	Accuracy
Coremark	201648	0.183467	117813	68069	9917	5849	0.6334
Compression	2638	0.242752	1697	204	435	302	0.8081
dna	15829	0.275925	12317	1202	1286	1024	0.8593
fft	3461	0.2075342	3056	46	6	353	0.8847
graph	5001	0.125468	4857	1	0	143	0.9712
mergesort	15444	0.225359	6797	6654	1533	460	0.5394
physics	54190	0.153606	24488	11037	16934	1731	0.7643

rsa	36049	0.113834	16197	14761	2671	2420	0.5234
sudoku	25606	0.120956	4369	16799	3830	608	0.3202

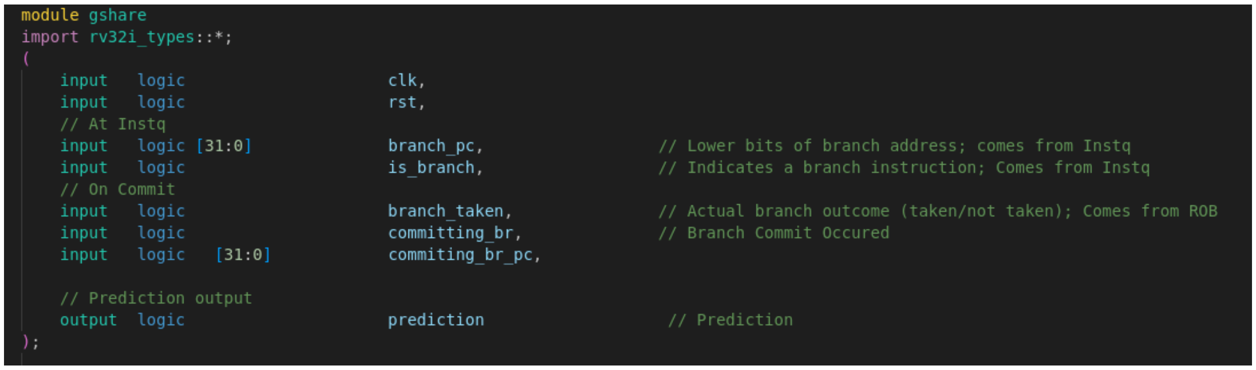
No Predictor

Test File	IPC
Coremark	0.140598
Compression	0.233766
dna	0.146967
fft	0.186059
graph	0.119631
mergesort	0.208658
physics	0.124592
rsa	0.107857
sudoku	0.242253

The Perceptron saw a lot of success on Coremark, Compression, dna, graph, and physics. This was due to most branches on specific pc values having a common pattern when deciding to branch/not branch. The predictor saw a slight performance decrease on the other test cases. This was due to most branches having irregular branching patterns.

Reference:
https://www.cs.cmu.edu/afs/cs/academic/class/15740-f18/www/papers/hpca01-jiminez-perceptron.pdf

Gshare Branch Predictor



```
module gshare
import rv32i_types::*;
(
    input   logic                   clk,
    input   logic                   rst,
    // At Instq
    input   logic [31:0]            branch_pc,              // Lower bits of branch address; comes from Instq
    input   logic                   is_branch,              // Indicates a branch instruction; Comes from Instq
    // On Commit
    input   logic                   branch_taken,           // Actual branch outcome (taken/not taken); Comes from ROB
    input   logic                   committing_br,          // Branch Commit Occured
    input   logic   [31:0]          commiting_br_pc,

    // Prediction output
    output  logic                   prediction               // Prediction
);
```

Another branch predictor that our group attempted was the Gshare Branch Predictor. Similar to our Branch Perceptron, we had a table of size 1024 to hold the unique state machines that pertained to different branch pcs. We also had a Global History Register of size 10 bits to assist in hashing into the Branch Table. In addition, each state machine was of size 2 bits: 2'b11 to represent STRONGLY_TAKEN, 2'b10 to represent WEAKLY_TAKEN, 2'b01 for WEAKLY_NOT_TAKEN, and 2'b00 for STRONGLY_NOT_TAKEN. The high-level overview of how my perceptron functions is as follows:

Advanced Design

1. After encountering a branch instruction, The bottom bits of the branch program counter (branch_pc[11:2]) were used to hash into my Branch Table. Using bits 11:2 instead of 9:0 was a design choice that I made because I noticed some indices were not being used due to the pc being multiples of 4. I XOR-ed the bottom 0 bits of the branch pc with the GHR to hash into the Branch table.
2. After fetching the state machine from the specified index, I used the 2 bits from the hashed index to figure out the current state of the branch instruction.
3. If the state machine was in either state STRONGLY_TAKEN or WEAKLY_TAKEN, I used the upper bit of the state machine (1'b1) to determine that the branch was taken. Else, if the current state is either WEAKLY_NOT_TAKEN or STRONGLY_NOT_TAKEN, I would once again use the upper bit (1'b0) to determine that the branch was not taken.
4. When the corresponding branch instruction is about to get committed, I will once again hash into the Branch Table by XOR-ing the branch pc with the current value of the GHR to see the original prediction for the given branch. I will then update each weight in the hashed index by incrementing each update by 1 the Branch was taken else decrement by 1. I will also set an upper bound at 2'b11 and a lower bound at 2'b00 to maintain having only 4 states in my state machine.
5. If a mispredict was determined on commit, I would relay back the branch pc + 3'b100 (4) to the fetch stage and combinational-ly set that as the next pc to make up for the branch mispredict by my Gshare predictor.

Design Tradeoffs:

Although the Gshare predictor was able to maintain a decently low clock period (2000) and the area did not increase as exponentially as that of the Perceptron (~21k), the accuracy did not match that of the Perceptron. Although we were able to maintain the clock period to be 2000, this was still a ~33% increase from our base clock period of 1450. Due to the combinational logic that was introduced with the Gshare predictor (Hashing into the Branch Table, identifying the current state, relaying the predicted pc the the Fetch module), we felt that the increase in clock period as well as the low accuracy was simply not make up for the 33% increase in clock period as well as the 21k increase in area. We decided to not add this branch predictor into our final design.

Performance Analysis:

Gshare Accuracy/speed up on varying test cases vs No Predictor:

Test File	Total Branch	IPC	TP	FP	TN	FN	Accuracy
Coremark	201648	0.166641	66260	37048	40938	57402	0.5316
Compression	2638	0.239510	1521	553	86	478	0.6091
dna	15829	0.223826	7792	1516	972	5549	0.5537
fft	3461	0.199532	2334	40	12	1075	0.6778
graph	5001	0.120930	3178	1	0	1822	0.6355
mergesort	15444	0.216979	4855	5651	2536	2402	0.4786
physics	54190	0.131947	15015	14840	13131	11204	0.5194
rsa	36049	0.113351	10202	10085	7347	8415	0.4868
sudoku	25606	0.120687	3676	16451	4178	1301	0.3067

No Predictor

Test File	IPC
Coremark	0.140598
Compression	0.233766
dna	0.146967
fft	0.186059
graph	0.119631
mergesort	0.208658
physics	0.124592
rsa	0.107857
sudoku	0.242253

The Gshare predictor saw a lot of success Compression, dna, fft, and graph. This was due to most branches on specific pc values having a common pattern when deciding to branch/not branch. The predictor saw a slight performance decrease on the other test cases. This was due to most branches having irregular branching patterns.

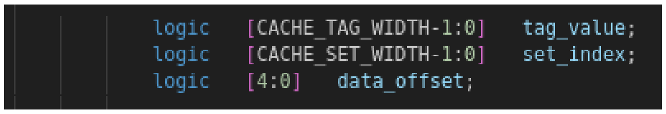
References:
https://people.cs.pitt.edu/~childers/CS2410/slides/lect-branch-prediction.pdf

Fully Parameterized Cache

Advanced Design:

This advanced feature involves parameterizing the cache set sizes. In other words, we have the flexibility to adjust cache set sizes to 16, 32, 64, or even larger, and the processor should function as if it were using the original cache configuration. To achieve this, we introduced a separate variable to store cache set sizes and replaced all hardcoded values related to set size with references to this variable. It's important to note that cache calculations are based on the address input, which is set to 32 bits. Therefore, increasing the number of sets corresponds to a decrease in tag value.

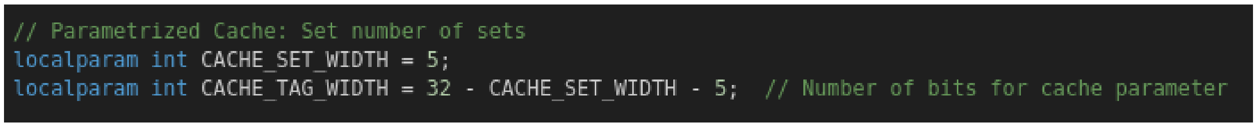
cache.sv



```
logic   [CACHE_TAG_WIDTH-1:0]   tag_value;
logic   [CACHE_SET_WIDTH-1:0]   set_index;
logic   [4:0]   data_offset;
```

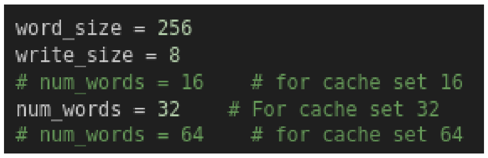
To experiment with different set configurations, we need to modify three files: the SRAM configuration data array file, the SRAM configuration tag array file, and the types.sv file. Subsequently, we must regenerate the SRAM. We need to update the Python script responsible for creating the SRAM to ensure that it accommodates the varying sizes of the data and tag arrays.

types.sv



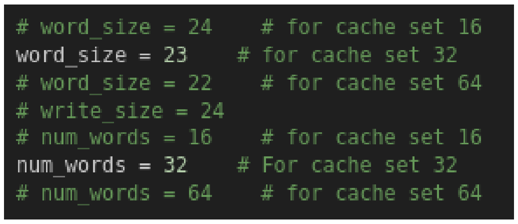
```
// Parametrized Cache: Set number of sets
localparam int CACHE_SET_WIDTH = 5;
localparam int CACHE_TAG_WIDTH = 32 - CACHE_SET_WIDTH - 5;  // Number of bits for cache parameter
```

mp_cache_data_array.py



```
word_size = 256
write_size = 8
# num_words = 16    # for cache set 16
num_words = 32    # For cache set 32
# num_words = 64    # for cache set 64
```

mp_cache_tag_array.py



```
# word_size = 24    # for cache set 16
word_size = 23    # for cache set 32
# word_size = 22    # for cache set 64
# write_size = 24
# num_words = 16    # for cache set 16
num_words = 32    # For cache set 32
# num_words = 64    # for cache set 64
```

Design Tradeoffs & Performance Analysis:

Verification of our cache implementation was conducted using the Verdi waveform viewer, where we manually inspected changes in cache sizes and accessed/saved data to specific sets. With 16 sets, set numbers should range from 0 to F, and with 32 sets, they should range from 0 to 1F. We tested two different set configurations: 16 and 32 sets. After an initial trial with 64 sets, we observed a significant increase in area utilization, leading us to abandon this configuration.

The table below presents IPC, area, and timing results for one of the test cases (coremarkIM.elf). An increase in IPC is evident when using 32 sets, as more data is stored in the cache, resulting in fewer cache misses and less data retrieval from Bmem. However, we did not observe a significant increase in IPC, and this observation could be attributed to several factors. One such factor pertains to our test cases. If there is limited data reuse within our test cases, it would result in fewer cache hits. Increasing the number of sets primarily enhances the probability of achieving a cache hit. The improvement in IPC comes at the expense of increased area and timing requirements, as we are allocating twice the amount of space compared to the 16-set configuration.

Fully Parameterized Cache (sets-only)	**IPC**	**Area**	**Timing**
16 sets	0.165549	161412	0.000211
32 sets	0.166747	208612	0.000069

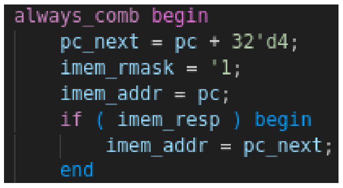
Pipelined Cache (Read-Only)

Advanced Design:

In the baseline cache model, we had a system where a single address could be processed within two cycles. Initially, the address is accepted in the idle state, then it moves into the compare_tag state for processing, and finally, the data is returned. This method revealed inefficiencies, as the processing of a subsequent address could only commence once the previous one had completed. To address this, we introduced a pipelined cache which enhances throughput by initiating the processing of a second address as soon as the first one has entered the compare_tag state. Essentially, while the baseline cache required two cycles to process each address, the pipelined cache can process an address per cycle, effectively doubling the throughput.

In the original baseline architecture, the fetch stage continuously broadcasted the program counter (PC) of the current address until the required data was retrieved. However, to optimize the system, we decided to alter the handling of addresses when data is not immediately available from the cache. Previously, we employed an iwait signal to continuously input the current PC into the imem_addr, but we have now shifted to using the imem_resp signal. This allows us to transition the imem_addr to pc_next combinationaly on the same cycle that the cache returns the data from the previous address.

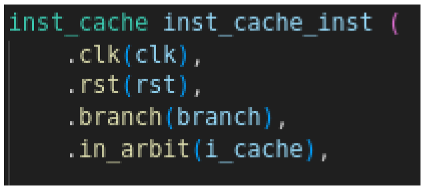
fetch.sv



```
always_comb begin
    pc_next = pc + 32'd4;
    imem_rmask = '1;
    imem_addr = pc;
    if ( imem_resp ) begin
        imem_addr = pc_next;
    end
```

Our focus has been on developing a read-only, pipelined cache specifically for instruction caching. To tailor the cache for this purpose, we cloned the existing cache and modified it accordingly.

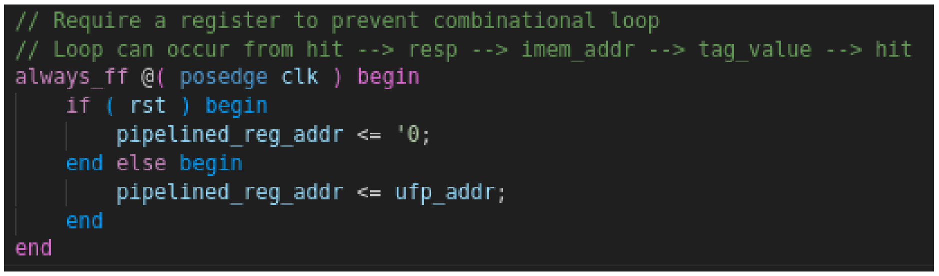
cpu.sv



```
inst_cache inst_cache_inst (
    .clk(clk),
    .rst(rst),
    .branch(branch),
    .in_arbit(i_cache),
```

One significant alteration was the elimination of the idle state return upon a cache hit. Instead, the system now cycles back to the compare_tag state, as the sequential nature of instruction processing assures that another instruction always follows. Moreover, to prevent combinational loops — an issue where the imem_resp signal could cause the imem_addr to immediately set to pc_next, thereby cyclically influencing the tag_value and hit determination — we introduced a latching mechanism to delay the address update until the next clock cycle.

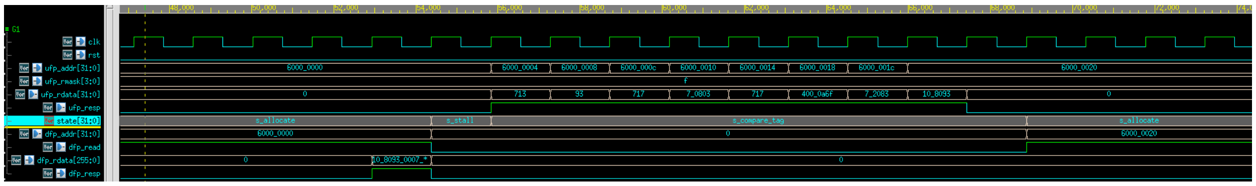
inst_cache.sv



```
// Require a register to prevent combinational loop
// Loop can occur from hit --> resp --> imem_addr --> tag_value --> hit
always_ff @( posedge clk ) begin
    if ( rst ) begin
        pipelined_reg_addr <= '0;
    end else begin
        pipelined_reg_addr <= ufp_addr;
    end
end
```

Design Tradeoffs & Performance Analysis:

Visualizing the operation of the pipelined processor through waveforms, it's apparent that address updates occur immediately upon receiving the response signal from the cache. The cache remains in the compare_tag state unless a miss occurs, at which point it transitions to the allocate state.



Performance metrics with coremark_IM.elf, contrasting configurations with 16 sets and 32 sets, were aimed to evaluate improvements. We anticipated a notable increase in IPC due to the new model's capability to feed more instructions each cycle, theoretically raising the IPC to as high as 1, as opposed to the baseline's maximum of 0.5. Surprisingly, we observed a slight decrease in IPC. Analysis indicated that with the new system, transitions into the allocate state became more frequent and extended, likely due to the higher rate of instruction processing. Initially, we suspected that the fetch stage might be unable to handle the increased bandwidth, but further investigation suggested that the bottleneck might reside within the CPU's functional unit, although this should not inherently limit IPC. Continued optimization are required to identify and address the precise causes of these performance discrepancies.

Fully Parameterized Cache (sets-only)	**IPC**	**Area**	**Timing**
16 sets	0.165033	161412	0.000211
32 sets	0.165820	207801	0.000127

Wallace Multiplier

Advanced Design:

The implementation of the Wallace Multiplier represents a significant departure from the conventional Shift-Add Multiplier, which often suffers from performance limitations. In our design, the Wallace Multiplier has been restructured into eight cycles, each corresponding to a reduction stage from the original 32-bit size. This reorganization significantly improves performance, as it theoretically reduces the time required to complete a multiply instruction. Each Wallace Stage in our design consists of pipeline registers storing AB partial products, carries, and sums, all of which are organized into arrays. The implementation process involved the use of a Python script to track the Wallace tree, facilitating the instantiation of Full and Half Adders and the assignment of outputs to subsequent Wallace stages. The key innovation lies in the utilization of the Wallace Algorithm, which optimizes the use of Full and Half Adders for each grouping of three rows, thereby maximizing efficiency and performance.

Design Tradeoffs:

Despite the considerable performance benefits offered by the Wallace Multiplier, there are tradeoffs to consider, primarily in terms of area utilization. The implementation of this advanced feature results in a significant increase in area requirements, primarily due to the large number of instantiated adders. Specifically, our design includes 906 Full Adders and 160 Half Adders, not including the generation of partial products. However, it's important to note that the performance gains achieved through the use of the Wallace Multiplier outweigh these tradeoffs. The improved efficiency and reduced execution time of multiply instructions contribute to overall processor performance, making the adoption of the Wallace Multiplier a worthwhile investment despite its associated area costs.

Performance Analysis:

Baseline:
Clock= 1450, IPC = 0.165549 , Area = 161412.508589, Timing = 0.000211

With Implementation:
Clock=1330, IPC = 0.193646 , Area = 224359.8177, Timing = 0.000007

Comparing the baseline performance metrics to those achieved with the implementation of advanced features, we observe notable improvements. The clock cycle time decreases from 1450 to 1330, indicating an increase in clock frequency. This aligns with the expected benefits of implementing advanced features such as the Wallace Multiplier, which can lead to faster execution times. Alongside the decrease in cycle time, the Instructions Per Cycle (IPC) significantly improves from 0.165549 to 0.193646, showcasing enhanced processor efficiency and throughput. However, these performance enhancements come at the cost of increased area utilization, with the area rising from 161412.508589 to 224359.8177.

DADDA Multiplier

Advanced Design:

In our processor design, the DADDA Multiplier presents a compelling alternative to the Wallace Multiplier, boasting the ability to produce multiplication results within a single cycle. Unlike the Wallace Multiplier, which utilizes a more aggressive reduction strategy, the DADDA Multiplier implements full and half adders conservatively, employing only the bare minimum necessary to advance to the next reduction stage. Additionally, the DADDA Multiplier utilizes more bits for the final addition due to its less aggressive reduction approach. Implementation of the DADDA Multiplier was facilitated by a Python script, which tracked the heights of the DADDA tree and the placements of each partial product. While this implementation outperforms the Shift Adder, it falls short compared to the Wallace Multiplier in terms of performance.

Design Tradeoffs:

Two significant tradeoffs arise with the implementation of the DADDA Multiplier. Firstly, since the implementation is purely combinational, it introduces a new critical path, potentially increasing the clock cycle and reducing frequency. This issue can be mitigated by pipelining the reduction stages, similar to previous approaches. Secondly, while the DADDA Multiplier utilizes fewer adders compared to the Wallace Multiplier—31 Half Adders and 899 Full Adders—it still incurs area costs. However, the reduced number of adders results in a lower area overhead compared to the Wallace Multiplier.

Performance Analysis:
Baseline:
Clock= 1450, IPC = 0.165549 , Area = 161412.508589, Timing = 0.000211

With Implementation:
Clock=3500, IPC = 0.196458 , Area =200704.9705, Timing = 0.000471

Comparing the performance metrics between the baseline and the implementation of the DADDA Multiplier reveals significant changes. Firstly, the clock cycle time increases substantially from 1450 to 3500, indicating a considerable decrease in clock frequency. This decrease in frequency may be attributed to the purely combinational nature of the DADDA Multiplier, which introduces a new critical path, resulting in longer cycle times. Despite the decrease in clock frequency, the Instructions Per Cycle (IPC) improve slightly from 0.165549 to 0.196458, indicating a modest enhancement in processor efficiency and throughput. However, it's important to note that this improvement in IPC comes at the cost of increased area utilization, with the area rising from 161412.508589 to 200704.9705. Additionally, the overall timing performance worsens, as reflected in the increase in timing from 0.000211 to 0.000471.

Conclusion

Overall, we successfully implemented a RISC-V Out-of-Order processor with Explicit Register Renaming. We gained significant knowledge in designing and verifying the correctness of our design. Additionally, we explored various advanced features, ranging from cache and branch prediction to the M-Extension. All these features broadened our understanding of optimization. Furthermore, we gained solid experience in different profiling techniques to analyze and evaluate our optimizations.

Our team enjoyed this project as it gave us insight into being computer engineers. At the start of our project, all of us were unemployed (thus, an unemployed trio). Now, we are all employed.