Riscalar

1st Bill Wang
Department of Computer Science and Engineering
Massachusetts Institute of Technology
Cambridge, MA, USA
wangbill@mit.edu

1st Catherine Tang
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
cattang@mit.edu

***Abstract*—We will build a pipelined superscalar RISC-V processor in SystemVerilog that is capable of out-of-order execution, branch prediction, and speculative execution. An FPGA is perfect for this project as it allows us to build a processor without the need to understand how to fabricate hardware, while simultaneously allowing us to easily experiment with different processor optimizations. Initially, we will build two baseline single-cycle and piplined processors capable of executing the standard RISC-V instruction set. Then we build and benchmark the superscalar processor, benchmarking the performance of our processor by analyzing the number of clock cycles and comparing with the baseline processor.**

***Index Terms*—Superscalar, RISC-V, Tomasulo's Algorithm, Data and Control Hazards, Tournament Branch Predictor, SystemVerilog**

I. Single Cycle Processor (Bill)

An overview of the entire single cycle processor is summarized Figure 6 within Appendix A. Given that the basic processor was not our project's focus, in the following subsections, we just elaborate on specific design choices.

A. Memory Design

We need to work with three classes of memory:

- **Instruction Memory:** We store our instructions in BRAM starting from address zero. Each instruction is 32 bits, so our BRAM has a width of 32 bits. Our BRAM has a depth of 1024 entries. Read and writes to the BRAM take two clock cycles. To make this work with single-cycle, we slowed our clock down by a factor of four. For the sake of the pipelined processor and the superscalar, we decided to use a single port write first memory because this allows load words that follow store words to read the updated value.
- **Register File:** Our registers are stored in a 2d logic array. The array is 32 by 32 since we have 32 registers, and each register holds 32 bits. Reads to our register file are combinational, while writes are sequential and take one clock cycle to complete.
- **Data Memory:** We also implement this in a BRAM of size 1024. Load words take two clock cycles to complete, but this is not a problem in the single-cycle because we slowed down our instruction period as described under Instruction Memory to account for this.

II. Pipelined Processor (Catherine)

Because the single cycle processor groups all of the logic into one clock cycle, the clock period is bottle necked by slow operations like memory reads and writes. Instead, we are able to achieve greater efficiency by pipelining our processor into the 5 stages. This allows for a shorter clock period, increasing throughput and lowering overall latency. Instead of handling only a single instruction at a time, our processor is now able to work on 5 instructions in flight.

We designed the pipelined processor as an intermediate between the baseline single-cycle processor and the superscalar, providing an additional more complex benchmark than our baseline single-cycle processor. With these additional features, the 5-stage pipelined processor has to handle data and control hazards. We describe them in the following subsections in brief because they are a subset of the hazards present in the superscalar.

A. Data Hazards

Because we are now handling multiple instructions in flight, we will begin processing subsequent instructions before previous instructions have finished executing. We may encounter read-after-write (RAW) hazards where an instruction's operands needs to read a register that is being written to by an in-progress operation. In these cases, we must stall the instruction in the instruction decode (ID) phase until the operand has been calculated or loaded. To implement this, we create ready signals for the stages ID, EXE, MEM, and WB. This allow us to stall by inserting NOPs (implemented as invalid instruction 32'b0) into the pipeline until the ready signal goes high. One further optimization we can add is to add in bypassing, so that an instruction that encounters a RAW hazard does not need to wait until the instruction it depends on is written to the register after WB and can instead receive the result as soon as it is calculated.

B. Control Hazards

Control hazards also arise when we start handling multiple instructions in flight. In particular, we do not know what the PC following a branch or jump instruction will be until the instruction completes the execute stage. One option would be to stall the entire pipeline until the branch instruction has executed, but this would be a large hit to our cycles-per-instruction (CPI). Instead, we use basic branch prediction,

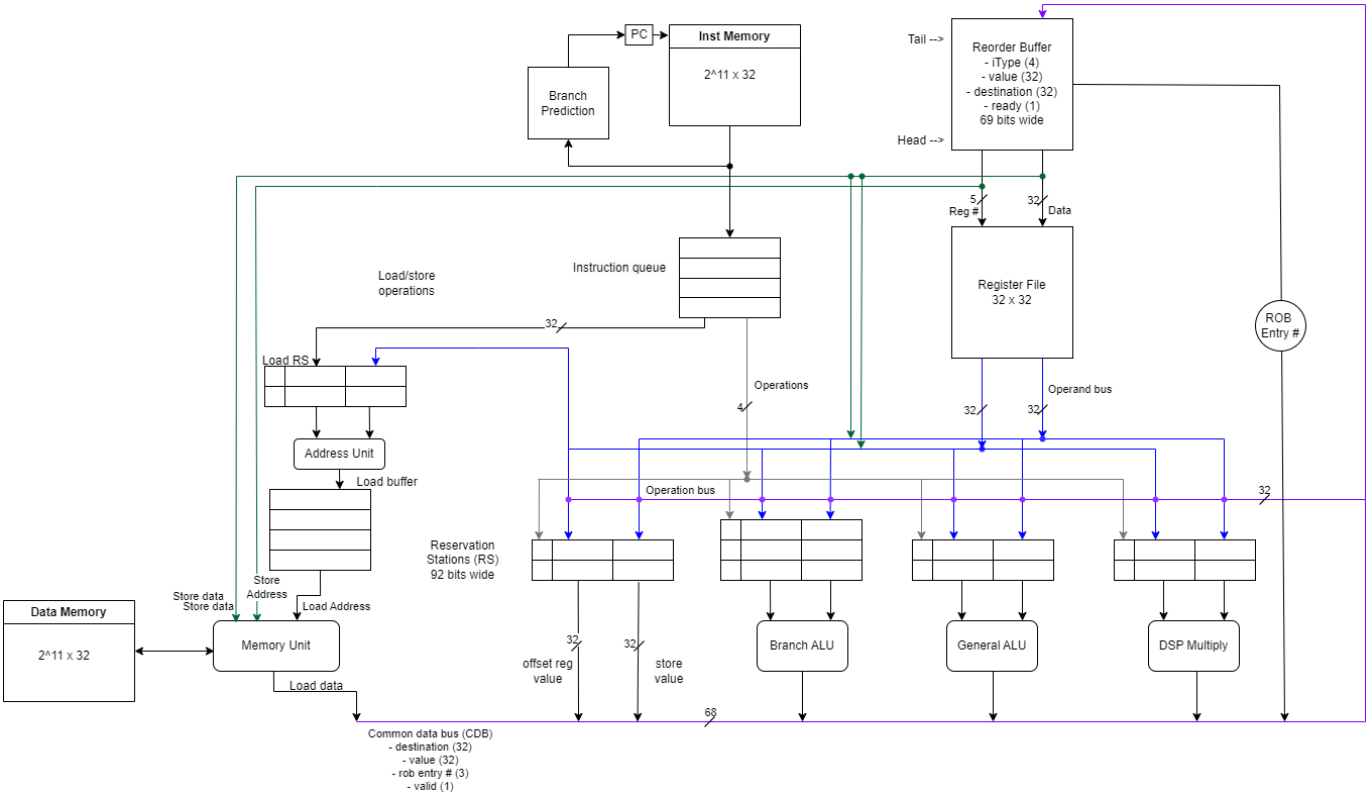

Fig. 1. Superscalar processor illustrating the key components of a superscalar processor including the functional units, the reorder buffer and reservation stations, and the memory units and register file. Figure based off diagram in Computer Architecture by Hennessy and Patterson.

assuming that all branches will not be taken at first and begin fetching instructions at PC+4 after the branch PC. Once we have the result of the branch, if we determine that the result is incorrect, we will replace the incorrectly fetched instructions with NOPs and begin fetching the instructions at the accurate PC.

III. Superscalar (Catherine)

This is the primary deliverable of our project. We implemented a pipelined and out-of-order superscalar processor through Tomasulo's algorithm. The diagram in Figure 1 summarizes all the components in the superscalar, described in greater detail in the following subsections. We will describe the components of the superscalar in the order of instruction flow through the processor.

A. Implementation of Tomasulo's Algorithm

1) Instruction Fetch: In the instruction fetch stage of the processor, instructions based on the program counter are read from the instruction BRAM, just as in the previous two processors. One consideration in both the pipeline and superscalar processor with multiple instructions in flight is aligning the PC with the instructions. One implementation challenge is the 2-cycle latency of the Xilinx BRAMs. But given that the BRAM is pipelined, the throughput should be 1 instruction per clock cycle, with just a 2 clock cycle delay. To overcome this, we fetch instructions two cycle ahead of when they are used. In other words, the address we read memory at is always 2 clock cycles ahead of the PC of the instruction fetched. We stall the PC for the first two clock cycles until the first memory result is ready, and then we start incrementing the PC while fetching the instruction at PC + 8. This leads to the complication where we need to know the PC 2 clock cycles ahead of time, which is not always the case when we have a branch or jump instruction. To overcome this, we stall for two clock cycles to read an instruction every time we branch, jump, or mispredict a branch. Although this does add complexity to the flow of instructions, we found this to be more performant than the alternative of waiting 2 clock cycles on every instruction fetch. One other consideration during the instruction fetch stage is if the rest of the superscalar (meaning the reorder buffer, reservation station, and instruction queue) is full, indicated by the `iq_ready` signal. In this scenario, we hold the PC constant and we wait until space opens up before continuing to fetch new instructions.

2) Branch Prediction and Decode: To fetch one instruction per clock cycle, we necessarily need to know whether the instruction fetched was a branch. As a result, we added a small branch decode that checks if the instruction is a branch and we then run branch prediction on the branches. We keep track of whether the branch was predicted as taken and we update the PC with the predicted address. The instruction and the decision to take or not take the branch is written into the

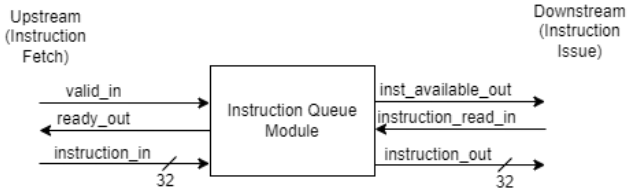

Fig. 2. Input and output wires for the instruction queue module. This instruction queue structure is similar to other queues we will implement including the functional unit results queue and the reorder buffer.

instruction queue.

3) Instruction Queue: Instructions fetched from BRAM (and for branches the prediction result) will be stored in this circular FIFO buffer. We need an instruction queue because there are situations where the next instruction cannot be issued yet because the reorder buffer (ROB) or the reservation stations (RS) are full. In this case, we want to continue fetching new instructions from instruction memory as to not waste clock cycles but we still need to keep track of our instructions that have yet to been dispatched (sent to a reservation station). In our instruction queue design, there are mutliple control signals going in and out of the instruction queue indicating the status of the instruction queue itself, as well as the components interacting with the queue on both ends as seen in Figure 2. On the side interfacing between the memory unit and the queue, there is a valid in input signal informing telling the buffer when the next input instruction is valid and should be stored in the queue. The queue also tells the upstream it is ready to receive another instruction while it still has room left in the queue. On the side where instructions are issued, the instruction queue has an instruction available signal letting the downstream logic know it can pull an instruction. There is also instruction read input signal that tells the queue its next instruction has been read and can now leave the buffer. In our implementation of this instruction queue, we decided to use a length 16 instruction queue as we thought this would give us ample buffer for instruction build up. Once an instruction reaches the front of the instruction queue, we wait for the the reservation station the instruction is assigned to (based on the instruction type) and the reorder buffer to be ready. Then we issue the instruction along with the reorder buffer entry number assigned by the ROB to this instruction.

4) Reservation Stations: We had one reservation station for each functional unit of our design (`rs_alu`, `rs_mul`, `rs_brAlu`, `rs_load`, `rs_store`). If an instruction has a data dependency that is not yet ready or the functional unit (FU) is in use, the instruction will be held in the functional unit's corresponding reservation station until the data dependency is resolved and the FU is free. As seen in Figure 3, each row in a reservation station has 9 fields: operation, rob entry number, Q_i, Q_j, V_i, V_j, i_{ready}, j_{ready}, busy. When an instruction is decoded and sent to the reservation station, if the instruction's operand is already known, the Q value is ignored, the V value is the value read from the register file, and the

4	3	3	3	32	32	1	1	1
OP	ROB#	Q_i	Q_j	V_i	V_j	i	j	B

Fig. 3. The reservation station above has a depth of 8 and a width of 80 bits. The breakdown between the 9 different fields is indicated by the bit indices. The subscripts i and j here correspond to register values 1 and 2. At any point, either Q or V for a given i and j will be 0. The i and j indicates if the operands are ready and B indicates that the row is busy.

operand ready bit (i_{ready} or j_{ready}) is 1. In the case where the instruction's operand is unknown, Q holds the re-order buffer entry number associated with the instruction that will return the result of the dependent operand. The ROB entry number comes from the register file, which maps each register to a ROB entry number if the instruction is still in-flight (in the ROB) and has yet to be committed. The busy bit indicates whether a row in the reservation station is free, or has an instruction currently in it.

5) Functional Units: Our superscalar has one ALU for general single cycle operations, a multiplier FU and a divider FU for multiply/divide instructions, a branch ALU for branch operations, and finally a memory unit to handle load/store instructions. The branch ALU is specialized to handle evaluating branch instructions. The reason to separate out this from the general ALU is because this allows us to get to branch instructions more quickly, which is crucial to save potentially wasted clock cycles due to control hazards. The multiple functional units are the core of what allows us to achieve instruction-level parallelism, giving the superscalar a signficant advantage. Here is a table of the number of clock cycles each functional unit takes to perform its calculation:

Functional Unit Latencies	
Functional Unit	Latency (clock cycles)
ALU	1
Branch ALU	0 (combinational)
Multiply	6
Memory Unit	2 (L) / 1 (S)

6) Load Buffer: After the operands of a load operation are known and the source address of the load operation is known, the instruction is moved into the load buffer first instead of directly loading from memory. This is important to handle a different flavor of hazards that are specific to load and store instructions. If a load instruction occurs after a store instruction, we need to check to avoid a RAW hazard, which occurs if both the load and the store happen at the same address. To determine if hazards exist, we connect the load buffer and the reorder buffer to check every pair of store and load instructions in flight if they meet the following two conditions to constitute a RAW hazard:

1) The store instruction occurs before the load.
2) The store instruction's address is unknown or known to be the same address as the load buffer.

Like the reservation station, instructions can be issued from the load buffer in any order once we confirm that the memory

unit is ready and there are no RAW data hazards.

7) Memory Unit: The memory unit is a module that wraps around the Xilinx BRAM that serves as our data memory. The memory unit ready and valid signal outputs to let the upstream modules know when an instruction can be issued and the downstream CDB when the output is available to be read. The memory unit takes in load instructions from the load buffer and store instructions from the reorder buffer. When concurrent loads and stores occur, we give priority to the store because the store takes only one clock cycle, and can allow more instructions to commit in the reorder buffer.

8) Common Data Bus: The common data bus (CDB) is a bus that connects to all reservation stations, the store buffer, and the reorder buffer (which then reaches the register file). Once a functional unit outputs data, the data is sent on the CDB, allowing whichever units that depend on the data to read the data directly. The idea behind a common bus is that all the hardware components share this wire which broadcasts information. This means that if multiple functional units need to use the bus, which in our case, happens not infrequently because we have a general purpose ALU and branch ALU which are 1 clock cycle and combinational respectively, there needs to be some way for one FU to grab the bus and send its results first. We considered widening the bus to allow for multiple functional units to write onto the bus at once. However, we found that this solution does not scale. Instead, we decided to use a priority-based selector that gives the bus to FU Alu, forcing FU AluBranch to stall, and then giving the bus to FU AluBranch in the next clock cycle. By comparing the ROB entry number of the result with the Q entries of the reservation stations, we can write the result into the value field of the dependent operands and into the reorder buffer.

9) Reorder Buffer: The reorder buffer (ROB) supports speculative execution by storing results of instruction executed out-of-order and committing them in order. The danger of speculative execution with out-of-order execution is that mis-predicted branches can change the state of the processor before the branches are evaluated. Instructions are placed in the ROB during the dispatch phase such that they are stored in the ROB in program order. As seen in Figure 4, each entry of the ROB has 4 fields: instruction type (ie. OP, OPIMM, MUL), resulting value of the instruction, destination (which could be an address or a register file index depending on the instruction type), and the ready flag, a bit indicating if the instruction has finished execution. The ROB has a head pointer which is the oldest instruction that is yet to be committed and the tail pointer, indicating the newest instruction added to the ROB. If during an issue phase, an operand value is calculated but still waiting to be committed into the register file, the operand value can be read from the ROB instead. The ROB commits the instruction at the head once it is ready. Commit is the point at which the system's state is modified by writing to the register file or memory. When a branch is speculatively taken incorrectly for example, the instructions that are executed do not change the state of the system yet. Instead, commits happen in order so only once the branch is evaluated do instructions following it

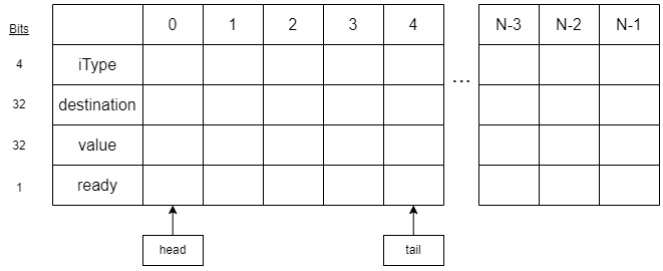

Fig. 4. A N-element reorder buffer is shown above. The reorder buffer ensures that the elements get committed in order even if executed out of order. The buffer is a circular FIFO buffer.

get considered. By then, we realize we have taken the wrong branch, so we flush the ROB and ignore all the instructions after the branch. One special case are store instructions, which change the system's state. Store instructions must be issued to the memory unit only once they are committed in the ROB because we do not want to change the system state out of order. To implement this ROB, we created a module that uses distributed RAM because we will frequently need to check and update the ROB every clock cycle. We decided to design our ROB to be 8 instructions deep, as we need to check the entire ROB every clock cycle which places a constraint on the length of the ROB. Each row has a width of

$$|\text{instType}| + |\text{val}| + |\text{dest}| + |\text{ready}| = 4 + 32 + 32 + 1 = 69 \text{ bits}$$

based on the 4 fields we described above. In choosing the length of our ROB, we found that the length of the ROB is a significant determinant of performance. If the ROB is too small, this is a bottleneck for the system and instructions will fill up in the instruction queue and no longer be able to be even fetched from memory.

B. Optimizations

1) Branch Prediction: This is a performance optimization allowing us to predict the next program counter right after an instruction is fetched. We implemented a tournament branch predictor based off the Alpha 21264 predictor. The branch predictor takes into consideration both local history (specific to the current branch pc) and global history (across all recent branches) to predict the next PC. However, to fit the hardware limitations, we had to modify the size of the buffers as well as the number of bits for our saturated counters. Figure 5 provides the specific details of our implementation.

2) Memory Optimizations: In the original Tomasulo superscalar, the reservation stations contain an additional 32 bit field where the address or the immediate can be stored. However, we were able to store the information we needed for each instruction type in other locations as to reduce memory overhead. For OPIMM and LOAD instructions, storing the immediate in the place of register value 2 was simple to do. For STORE instructions, in particular, we could not use the same technique because we had a rs1, rs2, and immediate field. We realized we could get around this by storing the immediate or offset temporarily in the destination field of the reorder buffer

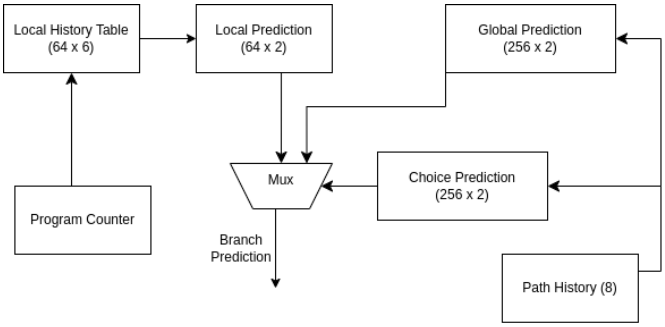

Fig. 5. Our implementation of the tournament branch predictor is shown above. It is based off the Alpha 21264 predictor.

entry for the instruction because anyways we only execute STORE instructions in order from the ROB. For BRANCH type instructions, we ran into the unique challenge of storing the decision of whether the branch was taken and storing the alternate PC. For example, if a branch was taken incorrectly, we need to be able to return back to pc + 4. Likewise, if a branch was predicted not taken, we would then need to know pc + immediate. We decided to store the alternate PC in the destination field of the branch and the value field of the branch was used to store the predicted take or not take value. Once the branch is evaluated, we are then able to compare with the original prediction and decide whether to use the backup PC or not.

IV. RISC-V (BILL)

A. RISC-V Toolchain

Riscalar is a RISC-V processor. RISC-V allows us to add custom instructions, and because we wanted the capability of easily doing that, we wrote our own Python RISC-V Toolchain. The toolchain is written in a way that's suited for easily adding in new instructions by separating instructions into groups.

B. RV32M Multiply Extension

Riscalar supports the RV32M multiply extension. See Appendix C for the ISA that our processor supports.

V. BENCHMARKING (BILL)

As hinted at previously, a superscalar processor excels under two main conditions: when there exists a lot of data dependencies, and when there exists operations that take a very long time. As we didn't implement floating point units (which are the primary operations where latency exists), there were only a select few instances where we saw notable speedup with a superscalar processor. As we want to give a full coverage of benchmarks, we will demonstrate those as well as other benchmarks where the superscalar does not do so well.

A basic set of varying ALU operations performed far worse on the superscalar than our pipelined processor.

Processor Benchmarks	
Size of Instructions	Superscalar Speed Up
Few ($\sim$ 10)	-70%
Medium ($\sim$ 20)	-70.2%
Large ($\sim$ 100)	-91%

For the first two tests, the superscalar processes around 1 instruction for every 3 instructions the pipelined processor processes. This makes sense. A pipelined processor with full bypassing is essentially optimized as much as possible to do ALU instructions, as the ALU will be kept busy every single clock cycle. Theoretically, if our processor was perfectly designed, the superscalar processor should match the pipeline processor's speed. However, since the hardware for the superscalar processor is so complicated, there were places where we needed to add in stalls as well as registers to meet timing. The specific modules will be talked more in the evaluation section, but the timing issues we encountered were the primary reason the superscalar performs poorly against pipelined.
Furthermore, we see that the efficiency of the superscalar decreased as instruction size increased. This is due to the fact that the reorder buffer and instruction queue had filled, so the pc is held constant and instructions were not being fed into the processor. Increasing those modules required careful planning, as size increases drastically, but mostly because the logic overhead becomes more complicated, resulting in timing issues once again.

A benchmark of multiplication and adds (including loads and stores to and from memory) show that a superscalar processor outperforms the pipelined processor for certain sizes. The pipelined processor currently just contains an ALU; we did not have time to add the multiplier, so the pipeline values were calculated by hand.

Processor Benchmarks	
Size of Instructions	Superscalar Speed Up
Few ($\sim$ 10)	6%
Medium ($\sim$ 20)	0.1%
Large ($\sim$ 100)	-22%

As above, large numbers of instructions are slower for the superscalar because control logic overhead increases when more instructions are in-flight, and because it was difficult increasing buffer sizes. However, since multiplication take 6 clock cycles, rather than one cycle as is the ALU, we were able to see some improvement in the superscalar performance due to parallelism (both from the add/multiplier, as well as the memory unit).

VI. DESIGN EVALUATION (BILL)

In the following table, we summarize the resource utilization of our program.

Resource Utilization	
Resource	Usage
BRAM	0.5%
DSPs	3%

Timing was a big issue for us, as it is for many superscalar designs, and it greatly affected our design. In particular, certain sections such as the load buffer and reorder struggled to meet timing requirements. Because of this, we needed to pipeline a lot of our designs, which resulted in a loss of latency. An alternative design we could have approached was using a slower clock in order to make more components combinational, but we decided against that approach because we weren't sure of the exact timing of each component.

We accomplished our minimum goal of building a working superscalar processor that supports integer instructions and RV32M, and completed the primary ideal goals of speculative execution and branch prediction; furthermore, we also built working versions of pipelined and single-cycle processors. Along the way, we also accomplished our tasks that we did not originally have on our checklist, such as building a toolchain from scratch to allow easy integration with official and custom extensions, and memory optimizations described above. Overall, we met our goals pretty well, and are happy with the design.

Aside from the obvious use case of processing machine code, Riscalar is flexible in that custom instructions can be easily added in. Because we wrote our assembler ourselves, we can easily fit our processor for any use case people may have. For instance, if the processor was being used in the context of cryptography, custom RISC-V keygen, signing/verification, and AES instructions can be inputted and decoded. IPs can be used for those modules, so most of the work would go in the decode and assembler section.

VII. Testing (Bill)

Unit and integration tests were deployed extensively for the superscalar processor. We show a few examples in the Appendix B.

VIII. Retrospective (Bill)

Looking back, there were many things we could have done better:

1) A lot of our logic occurred in the top_level file. This was partially unavoidable as the modules need a center location to interact with each other; however, there were many components we could have modularized and tested individually to ease the debugging process. In particular, the number of variables neccesary to debug the entire process as a whole every time grew large very quickly, so establishing a consistent naming scheme and grouping modules into larger modules would have helped.
2) We built three different processors – single-cycle, pipelined, and superscalar – to aid with benchmarking. In retrospect, we should have began on the superscalar earlier, as it's design was a significant change from the earlier pipelined and single-cycle processors.
3) One of the main reasons Riscalar didn't show significant improvement over our pipelined processor is because our functional units were not designed for long latency instructions. In particular, we didn't have any floating point operations, which are one of the main reasons superscalar architecture is advantageous. Looking back, we should have made it a goal to support floating point operations, and although it would have been a somewhat easy fix, by the time we realized, we had other issues we needed to deal with first.

APPENDIX A
PROCESSOR BLOCK DIAGRAMS

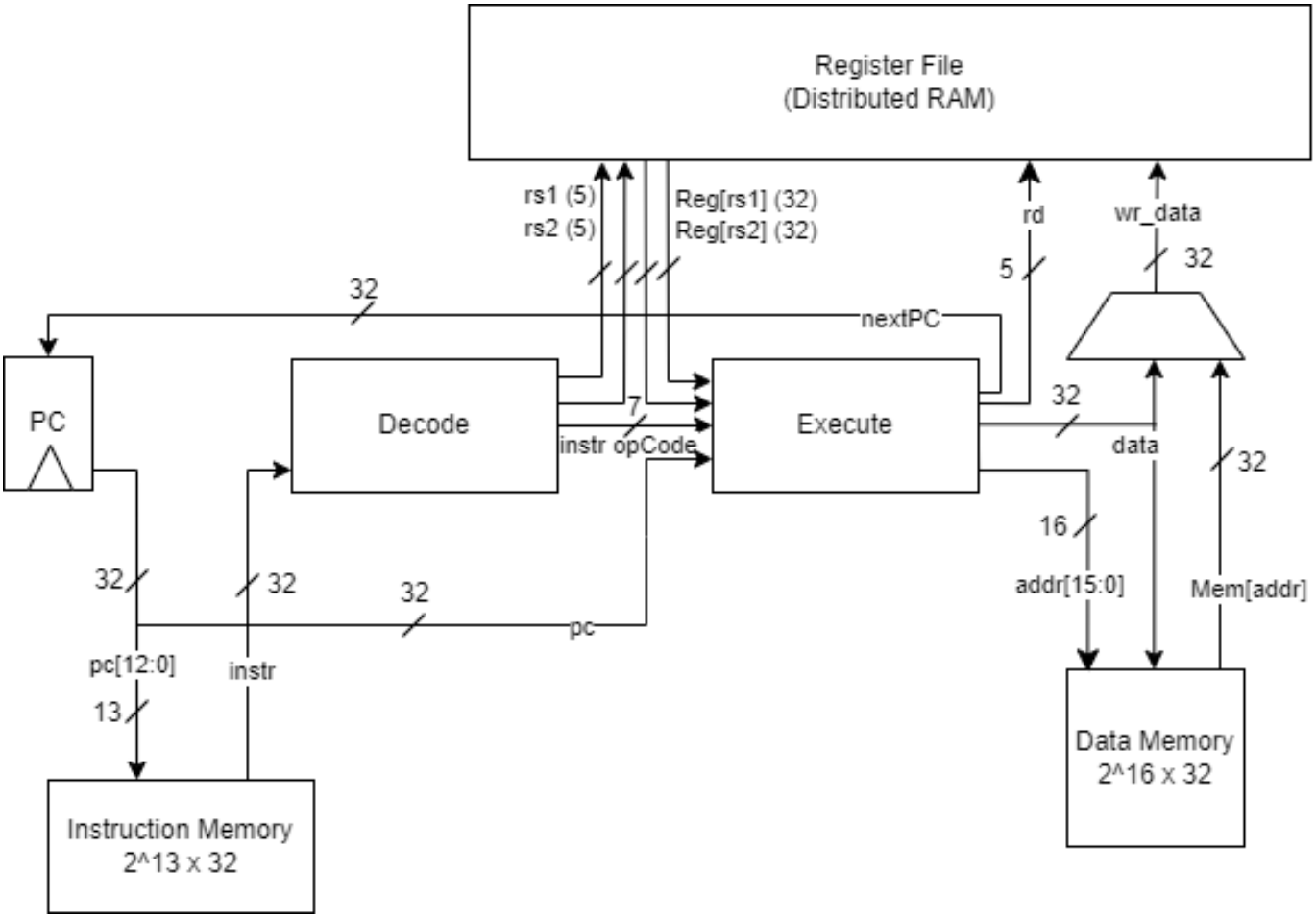

Fig. 6. A Single Cycle Processor with 5 stages. An instruction moves through all 5 stages before the next instruction is loaded in the following clock cycle. Figure based off 6.004 Lecture Notes.

APPENDIX B
SECOND APPENDIX

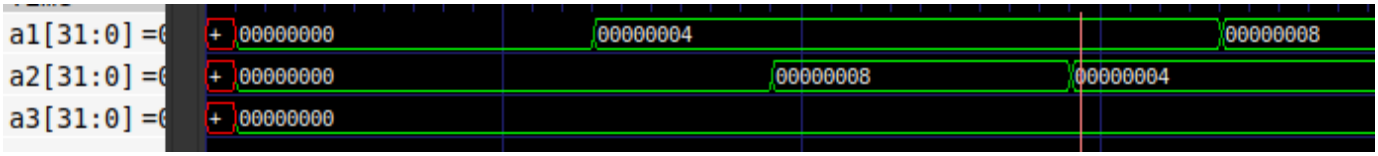

Fig. 7. GTKWave output of the Superscalar processor. Tests the swapping of load and store between registers. Basic test that requires no data dependencies. The instructions are listed below.

```
addi a1, a1, 4
addi a2, a2, 8
sw a1, 0(x0)
sw a2, 4(x0)
lw a2, 0(x0)
lw a1, 4(x0)
```

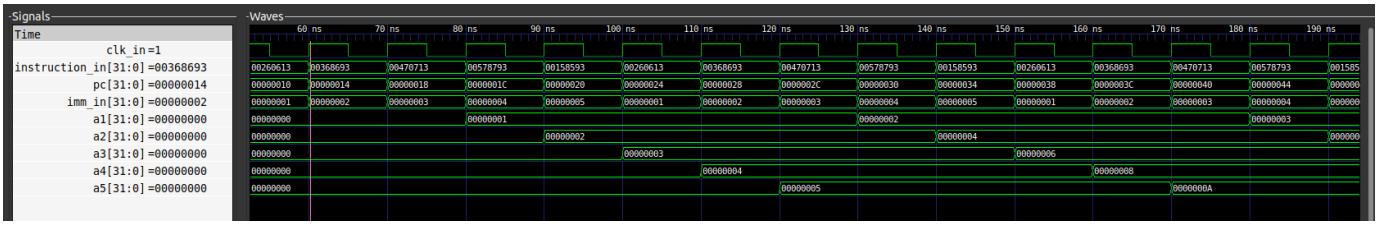

Fig. 8. GTKWave Output of a test of unbypassed 5-stage pipelined processor. There are no data dependencies and therefore no bypassing needed but we can see a result being generated and written to the register every clock cycle. The instructions are listed below.

```
addi a1, a1, 1
addi a2, a2, 2
addi a3, a3, 3
addi a4, a4, 4
addi a5, a5, 5
addi a1, a1, 1
addi a2, a2, 2
...
```

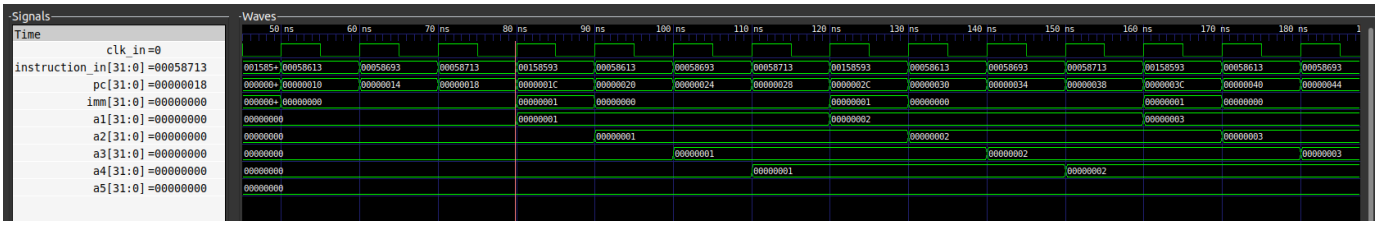

Fig. 9. In this test of the pipelined processor (assembly code below), we test full-bypassing between each of the EXE, MEM, and WB stages. We see that the only way for the 2nd, 3rd, and 4th instructions to get their results is to use bypassing, because the value of a_1 has not reached the register yet.

```
addi a1, a1, 1
addi a2, a1, 0
addi a3, a1, 0
addi a4, a1, 0
...
```

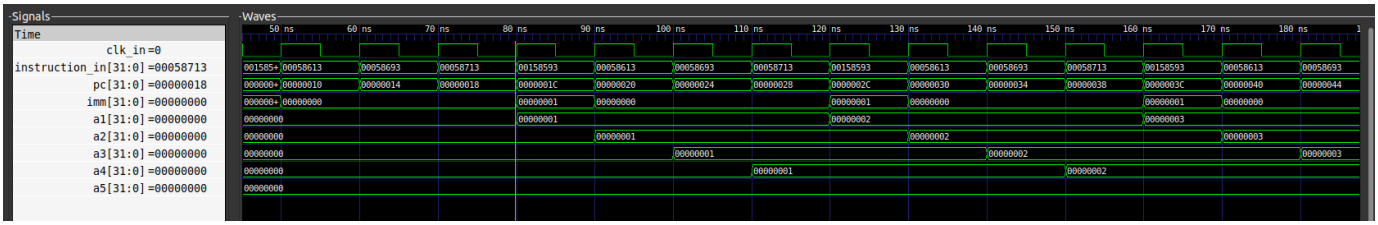

Fig. 10. In this test of the pipelined processor (assembly code below), we test full-bypassing between each of the EXE, MEM, and WB stages. We see that the only way for the 2nd, 3rd, and 4th instructions to get their results is to use bypassing, because the value of a_1 has not reached the register yet.

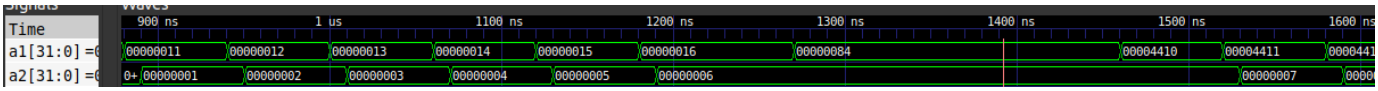

Fig. 11. Here we tested the superscalar with multiple different types of instructions as well as filling the reservation station and reorder buffer.

```
lw a1, 0(x0)
mul a1, a1, a1
addi a1, a1, 1
addi a2, a2, 1
addi a1, a1, 1
addi a2, a2, 1
addi a1, a1, 1
addi a2, a2, 1
addi a1, a1, 1
addi a2, a2, 1
addi a1, a1, 1
addi a2, a2, 1
addi a1, a1, 1
addi a2, a2, 1
mul a1, a1, a2
sw a1, 0(x0)
lw a1, 0(x0)
mul a1, a1, a1
addi a1, a1, 1
addi a2, a2, 1
...
```

Standard Extensions

RV32M Multiply Extension

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)
`mul`	MUL	R	`0110011`	`0x0`	`0x01`	`rd = (rs1 * rs2)[31:0]`
`mulh`	MUL High	R	`0110011`	`0x1`	`0x01`	`rd = (rs1 * rs2)[63:32]`
`mulsu`	MUL High (S) (U)	R	`0110011`	`0x2`	`0x01`	`rd = (rs1 * rs2)[63:32]`
`mulu`	MUL High (U)	R	`0110011`	`0x3`	`0x01`	`rd = (rs1 * rs2)[63:32]`
`div`	DIV	R	`0110011`	`0x4`	`0x01`	`rd = rs1 / rs2`
`divu`	DIV (U)	R	`0110011`	`0x5`	`0x01`	`rd = rs1 / rs2`
`rem`	Remainder	R	`0110011`	`0x6`	`0x01`	`rd = rs1 % rs2`
`remu`	Remainder (U)	R	`0110011`	`0x7`	`0x01`	`rd = rs1 % rs2`

RISC-V Reference

RISC-V Instruction Set

Core Instruction Formats

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 Integer Instructions

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
add	ADD	R	0110011	0x0	0x00	rd = rs1 + rs2	
sub	SUB	R	0110011	0x0	0x20	rd = rs1 - rs2	
xor	XOR	R	0110011	0x4	0x00	rd = rs1 ^ rs2	
or	OR	R	0110011	0x6	0x00	rd = rs1 | rs2	
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2	
sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 << rs2	
srl	Shift Right Logical	R	0110011	0x5	0x00	rd = rs1 >> rs2	
sra	Shift Right Arith*	R	0110011	0x5	0x20	rd = rs1 >> rs2	msb-extends
slt	Set Less Than	R	0110011	0x2	0x00	rd = (rs1 < rs2)?1:0	
sltu	Set Less Than (U)	R	0110011	0x3	0x00	rd = (rs1 < rs2)?1:0	zero-extends
addi	ADD Immediate	I	0010011	0x0		rd = rs1 + imm	
xori	XOR Immediate	I	0010011	0x4		rd = rs1 ^ imm	
ori	OR Immediate	I	0010011	0x6		rd = rs1 | imm	
andi	AND Immediate	I	0010011	0x7		rd = rs1 & imm	
slli	Shift Left Logical Imm	I	0010011	0x1	imm[5:11]=0x00	rd = rs1 << imm[0:4]	
srli	Shift Right Logical Imm	I	0010011	0x5	imm[5:11]=0x00	rd = rs1 >> imm[0:4]	
srai	Shift Right Arith Imm	I	0010011	0x5	imm[5:11]=0x20	rd = rs1 >> imm[0:4]	msb-extends
slti	Set Less Than Imm	I	0010011	0x2		rd = (rs1 < imm)?1:0	
sltiu	Set Less Than Imm (U)	I	0010011	0x3		rd = (rs1 < imm)?1:0	zero-extends
lb	Load Byte	I	0000011	0x0		rd = M[rs1+imm][0:7]	
lh	Load Half	I	0000011	0x1		rd = M[rs1+imm][0:15]	
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]	
lbu	Load Byte (U)	I	0000011	0x4		rd = M[rs1+imm][0:7]	zero-extends
lhu	Load Half (U)	I	0000011	0x5		rd = M[rs1+imm][0:15]	zero-extends
sb	Store Byte	S	0100011	0x0		M[rs1+imm][0:7] = rs2[0:7]	
sh	Store Half	S	0100011	0x1		M[rs1+imm][0:15] = rs2[0:15]	
sw	Store Word	S	0100011	0x2		M[rs1+imm][0:31] = rs2[0:31]	
beq	Branch ==	B	1100011	0x0		if(rs1 == rs2) PC += imm	
bne	Branch !=	B	1100011	0x1		if(rs1 != rs2) PC += imm	
blt	Branch <	B	1100011	0x4		if(rs1 < rs2) PC += imm	
bge	Branch ≥	B	1100011	0x5		if(rs1 >= rs2) PC += imm	
bltu	Branch < (U)	B	1100011	0x6		if(rs1 < rs2) PC += imm	zero-extends
bgeu	Branch ≥ (U)	B	1100011	0x7		if(rs1 >= rs2) PC += imm	zero-extends
jal	Jump And Link	J	1101111			rd = PC+4; PC += imm	
jalr	Jump And Link Reg	I	1100111	0x0		rd = PC+4; PC = rs1 + imm	
lui	Load Upper Imm	U	0110111			rd = imm << 12	
auipc	Add Upper Imm to PC	U	0010111			rd = PC + (imm << 12)	
ecall	Environment Call	I	1110011	0x0	imm=0x0	Transfer control to OS	
ebreak	Environment Break	I	1110011	0x0	imm=0x1	Transfer control to debugger	