**Custom Central Processing Unit – A Case Study for Small Scale, Pipelined Computing Architecture with Out-of-Order Execution, In-Order-Commit**

By: Joseph Post

**Purpose:** In this paper, a custom, soft microprocessor Central Processing Unit (CPU) is designed, developed, tested, and analyzed. The objective of this CPU is to execute instructions out-of-order (OOO), commit instructions in-order, and utilize predictive branch execution and a 5-stage pipeline. The CPU utilizes a 16-bit architecture with a Harvard memory scheme. The CPU supports simple input/output (IO) instructions in the form of 16 digital inputs and 16 digital outputs, as well as an integrated circuit (I2C) interface.

The high-level architecture of the CPU is as follows:

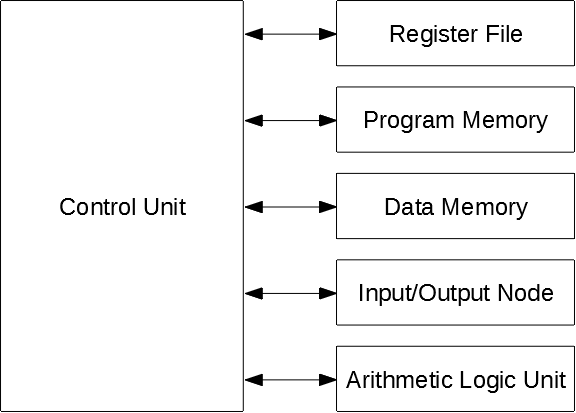


Figure 1: High Level System Architecture

**Abstract:** In order to accomplish the goals of executing instructions out-of-order (OOO), committing instructions in-order, and utilizing predictive branch execution, the following architectural features have been incorporated:

1) Out-of-order execution: A pre-fetch buffer, or Look Ahead Buffer (LAB), is utilized in the first pipeline stage.

2) In-order commits: A Re-Order Buffer (ROB) is utilized to maintain program instruction order and track those instructions which are speculatively executed.

3) Predictive branch execution: The CPU implements a simple predictive branch execution algorithm of assuming the branch condition is **not** met, and simply fetching the next Program Memory (PM) instruction.

This CPU does not utilize any data busses, in the classical sense, and does not utilize the status register tracking zero, carry, negative, or overflow status of arithmetic operations. This design decision was chosen because the architecture was designed to execute instructions out-of-order, making a status register unreasonably complicated since it would have to track any particular instruction’s ALU results to be able to execute a subsequent instruction which relied on those results. A two- and three-bus architecture was explored briefly, which utilized a signal arbitration module, but the complexity of this arbitration module did not outweigh the simplicity of simply forwarding data between the necessary components; although this implementation would have potentially allowed forwarding of IO read results.

At a high level, to read data from a particular block, the CU will dispatch an address and control signals to each applicable block (see Figure 1), and each applicable block will forward data to a destination block. For example, for the IO Peripheral block, this block will simply return all input states, all output states, or an I2C message. For the RF block, this block will output the data of a particular register. For the Memory block, this block will output the applicable data from the PM, or DM. For write operations, the process is almost identical with the exception that various write\_enable signals will be set to establish the proper write permission for a particular operation.

A block diagram for each block in Figure 1 is shown below for clarity.

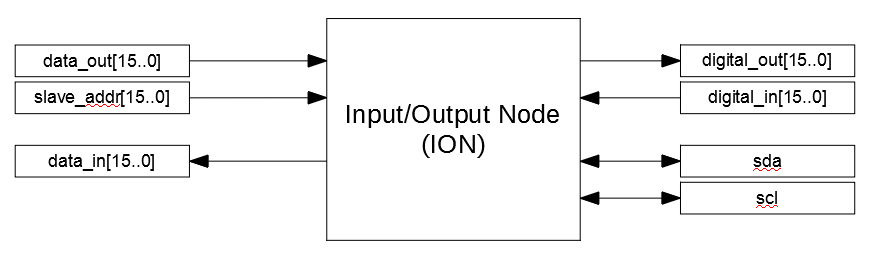


Figure 2: IO Node (ION) High-Level Block Diagram

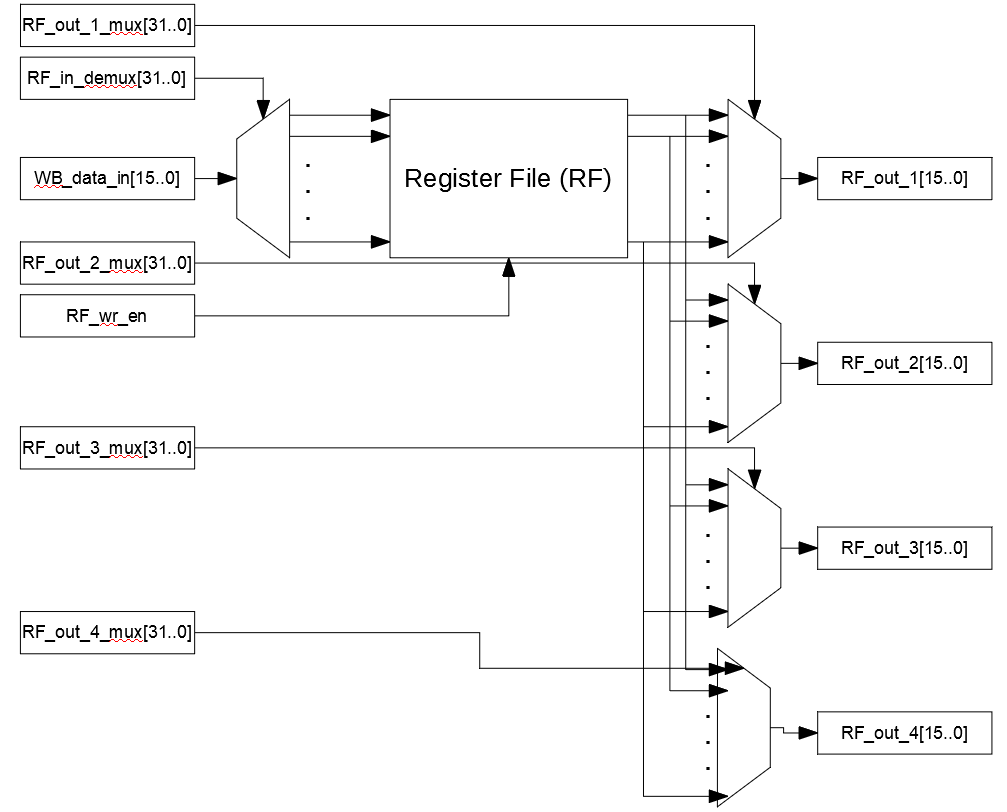


Figure 3: Register File (RF) Block Diagram

The RF block contains 32 registers, similar to the classic Reduced Instruction Set Computer (RISC) architecture. Therefore, 5 bits will be required to represent a register in the RF. Each register has a 'valid’ bit that is set to ‘0’ when an instruction writing to that register is issued, and is set to ‘1’ when the result of that operation is committed from the ROB back to the RF.

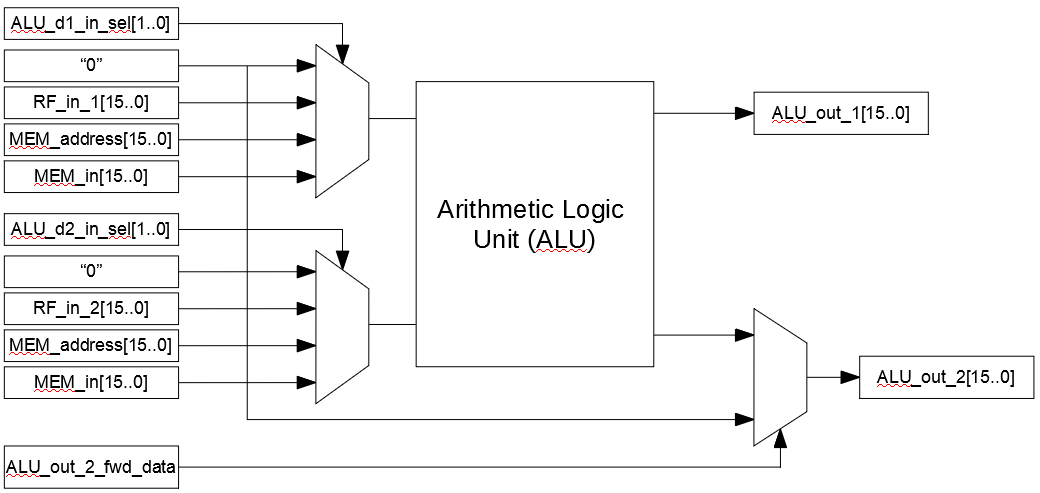


Figure 4: Arithmetic Logic Unit (ALU) Block Diagram

**The Instruction Set Architecture (ISA):** This is a 16-bit architecture, and, as such, has 16-bit instruction words (IWs). The general structure of the IW begins with a 4-bit Operational Code (opcode), followed by one or two register references, and atwo optional Instruction Select (inst\_sel) bits.

|  |  |  |
| --- | --- | --- |
| Table 1: Instruction Set Architecture (ISA) Summary | | |
| ADD (I) | 0000 | Add (immediate) |
| SUB (I) | 0001 | Subtract (immediate) |
| MULT (I) | 0010 | Multiply (immediate) |
| DIV (I) | 0011 | Divide (immediate) |
| LOG | 0100 | Logic functions (x4) |
| RT(L:R)(I) | 0101 | Rotate (immediate) |
| S(L:R)L (I) | 0110 | Shift logic (immediate) |
| S(L:R)A (I) | 0111 | Shift arithmetic (immediate) |
| LD/ST | 1000 | Load/store from/to DM |
| JMP | 1001 | Jump immediate to address |
| BNE(Z) | 1010 | Branch if not equal (to zero) |
| IOW(R) | 1011 | Write/read to digital outputs/inputs |
| LOGI | 1100 | Logic functions using immediate values |
| CP | 1101 | Copy reg2 to reg1 |
| - | 1110 |  |
| - | 1111 |  |

The two unused opcodes may be utilized in a future revision to the ISA. The 4-bit opcode leaves 12 bits available for the remainder of the IW. These bits depend entirely on the operation being performed. All possible operations, with enumeration of the IW, are shown below. Note that Register 1 is also the destination of the operation. This is a potential con of using a 16-bit bit uA, since the classic RISC architecture typically includes a third register in the IW for greater flexibility; however, this simplicity precludes the general need for a reserved ‘zero’ register.

This highlights that there must be a balance between the ISA, number of available operations in hardware, which operations can be emulated during code compilation (more on this later), and RF space. Fewer instructions in the ISA and fewer registers in the RF would allow a third register (e.g., Reg3) in the IW; however, utilizing a 32-register RF and providing more common instructions in the ISA will provide most of the operational flexibility required to handle simple to medium-complexity tasks. The actual IW composition for each instruction is as follows.

ADD\*, SUB\*, MULT\*, DIV\*, SF(L:R)L\*\*, SF(L:R)A\*\*, ROT\*\*, LOG\*\*\*:

|  |  |  |  |
| --- | --- | --- | --- |
| OpCode  (4 bits) | Reg1  (5 bits) | Reg2  (5 bits) | Inst\_sel\*  (2 bits) |

\*See Table 2 for the inst\_sel for these instructions.

\*\*See Table 3 for the inst\_sel implementation for these instructions.

\*\*\*See Table 4 for the inst\_sel implementation for this instruction.

MSB of immediate value dictates shift direction (1 = left, 0 = right). This leaves 25 = 32 shift values, which is obviously more than an entire IW. Therefore this is enough for all shift purposes.

|  |  |
| --- | --- |
| Table 2: Inst\_sel for ADD, SUB, MULT, DIV Operations | |
|  | Description |
| 00 | Use Reg2 for reg-reg operation |
| 01 | Use "Reg2" field as immediate value |
| 10 | (unused) |
| 11 | (unused) |

|  |  |  |
| --- | --- | --- |
| Table 3: Inst\_sel for S(L:R)A(I), S(L:R)L(I), and RT(L:R)(I) Operations | | |
|  | Description | Reg1 / Reg2 |
| 00 | Shift/rotate left | Dest Register / Value from Reg\* |
| 01 | Shift/rotate right | Dest Register / Value from Reg\* |
| 10 | Shift/rotate left immediate | Dest Register / Immediate value\*\* |
| 11 | Shift/rotate right immediate | Dest Register / Immediate value\*\* |

\*Use the contents in Reg2 to shift the contents of Reg1.

\*\*Use Reg2 as an immediate value to shift Reg1.

|  |  |
| --- | --- |
| Table 4: Inst\_sel for LOG(I) Operations | |
|  | Description |
| 00 | AND |
| 01 | OR |
| 10 | XOR |
| 11 | NOT (the "Reg 2" field is unused for this operation) |

LD, ST:

|  |  |  |  |
| --- | --- | --- | --- |
| OpCode (4 bits) | Reg1  (5 bits) | Reg2  (5 bits) | Inst\_sel  (2 bit) |

|  |  |  |
| --- | --- | --- |
| Table 5: Inst\_sel for Loads/Stores | | |
|  | Description | Reg1 / Reg2 |
| 00 | LD | Dest Register\* / Memory Offset from Reg\*\* |
| 01 | LD | Dest Register\* / Memory Offset value\*\*\* |
| 10 | ST | Src Register\* / Memory Offset from Reg\*\* |
| 11 | ST | Src Register\* / Memory Offset value\*\*\* |

\*This is the register who will be the destination of a load, or the source register for a store.

\*\*Use the contents in Reg2 to offset the next IW memory address to compute the effective memory address.

\*\*\*Use Reg2 as immediate value by which to add to the next IW in the PM. This can be set to "0" to effectively just use the next IW as the effective address.

Given the DM address scheme, there will be 211 = 4,096 total DM addresses available. This should provide adequate operational flexibility. The I2C interface and any higher-level implementations of a stack and heap will demand a fair portion of the available DM space, and can be loaded into DM memory before use of the CPU via pre-constructed Memory Initialization File (MIF).

BNE(Z):

|  |  |  |  |
| --- | --- | --- | --- |
| OpCode (4 bits) | Reg1  (5 bits) | Reg2  (5 bits) | Inst\_sel  (2 bit) |

|  |  |  |
| --- | --- | --- |
| Table 6: Inst\_sel for Branches | | |
| BAM | Description | Reg1 / Reg2 |
| 00 | Branch if not zero | Evaluate Reg1 to branch if not zero |
| 01 | Branch if not equal | Compare Reg1 to Reg2, branch if equal |
| 10 | (unused) | - |
| 11 | (unused) | - |

IOW(R):

|  |  |  |  |
| --- | --- | --- | --- |
| OpCode (4 bits) | Reg1  (5 bits) | Reg2  (5 bits) | Inst\_sel  (2 bit) |

|  |  |  |
| --- | --- | --- |
| Table 7: Inst\_sel for ION Instructions | | |
| IOS | Description | Reg1 / Reg2 |
| 00 | GPIO read | Destination register / unused |
| 01 | GPIO write | Source register / unused |
| 10 | I2C read | Destination register / Slave address (only 5 bits) |
| 11 | I2C write | Source register / Slave address (only 5 bits) |

JMP:

|  |  |  |
| --- | --- | --- |
| OpCode  (4 bits) | Jump Address  (11 bits) | (Unused)  (1 bit) |

CP:

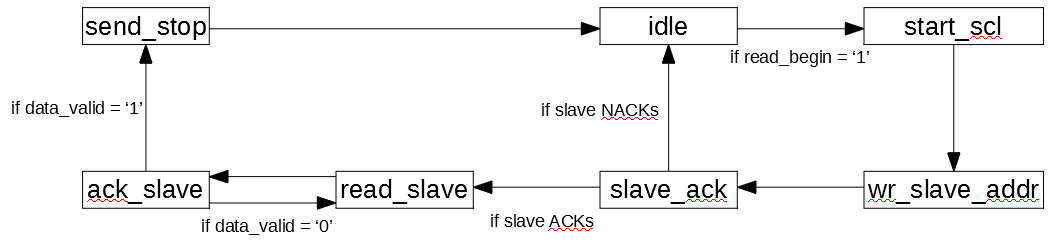
|  |  |  |  |
| --- | --- | --- | --- |
| OpCode (4 bits) | Reg1  (5 bits) | Reg2  (5 bits) | (Unused)  (2 bits) |

Copies content from reg2 into reg1 by adding zero with the reg2 content, and committing to reg1.

Note that the first address entering the RF, Addr\_1, is used to select which register to write to, as well as which register to read to RF\_out\_1. This RF implementation is fairly multiplexer (mux) intensive, which could cause fanout issues; however, given the simple implementation of muxes (i.e., lookup tables (LUTs)), they will likely not be a significant burden to the overall design, as they are not used to propagate clock signals and will be primarily static during most operations.

**Detailed Block Descriptions**

**ION:** The ION is capable of interfacing with a single I2C module, as well as reading 16 digital inputs and writing and reading 16 digital outputs. A dedicated opcode for IO operations is implemented. The I2C interface is capable of reading a single byte from a slave device and writing a single byte to a slave device. The overall flowchart for I2C read and write operations is as follows. Each node represents a VHDL state for that particular operation:

Figure 5: I2C Read Operation Flow Chart

Note that the I2C module will re-request data if any bit transmitted was invalid, and will ACK and stop communications upon the first valid byte of data received. Conversely, the overall flowchart for I2C write operations is as follows:

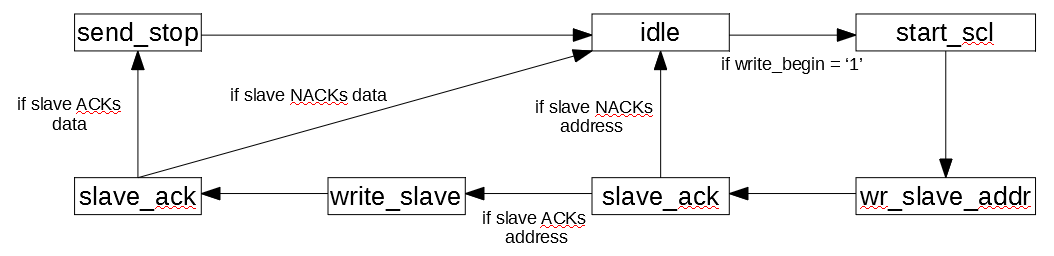


Figure 6: I2C Write Operation Flow Chart

Note that the I2C module will only write a single byte of data before stopping communication. Subsequent writes will require another transmission.

**Control Unit (CU):** The CU produces all control signals, directs the flow of data, and overall operational execution of the CPU. Since the CPU will be pipelined and allow OOO execution, the CU requires several pipeline registers as well as an in-order commit buffer to ensure that data is committed in accordance with the program sequence. The purpose of OOO execution is to ensure that all available portions of the CPU are utilized as often as possible to provide the lowest cycles per instruction (CPI), which would be as low as 1 for a theoretically perfect processor. The CPU operates using a 5-stage pipeline as follows:

Instruction Fetch → Instruction Decode → Execute → Memory Access → Write Back

The CU consists of a child CU block for each pipeline stage. Each child CU block is capable of requesting data from various blocks in the architecture, as previously mentioned.

**Instruction Fetch (IFetch) Stage:** The IFetch stage interfaces with the PM and is capable of issuing PM instructions directly to the pipeline, or will buffer them as discussed below.

The IFetch stage Look Ahead Buffer (LAB) is a 5-entry, first-in, first-out (FIFO) buffer which stores all non-issued PM IWs. The LAB searches from the bottom-up to determine whether a particular instruction poses any control or data hazard as described above. This is accomplished using two std\_logic\_vectors: 1) a pipeline data hazard indication (PL\_datahaz[4..0]) and 2) and 2) a LAB data hazard indication (LAB\_datahaz[4..0]). The latter determines whether any LAB instruction poses a data hazard to any instruction below it in the LAB, and the former indicates whether any LAB instruction requires data from an instruction currently in the pipeline. Herein lies the implementation of the OOO execution of the CPU.

A PM\_datahaz std\_logic signal is used to identify whether the incoming PM IW poses any data hazard to the LAB or pipeline. If ‘0’, the IFetch can issue this instruction. Whether the PM IW or a LAB IW, the issued IW is sent to the Instruction Decode stage.

Note that any instruction relying on the current ALU output will not be flagged as a data hazard, since this information can be forwarded back to the ALU, thereby providing data forwarding capability. This CPU does not fully forward data (I.e., forward data from every pipeline stage to every other pipeline stage), given the unnecessary complexity that this entails. Forwarding ALU output data back in to the ALU allows any two data-dependent instructions to be executed sequentially, delayed by a mere clock cycle.

**Instruction Decode (ID) Stage:** This stage decodes the registers used in the IW issued from the IFetch stage. It issues RF output mux signals and forwards several data forwarding signals to the Instruction Execute stage, as well as the IW itself.

**Instruction Execute (EX) Stage:** This stage decodes the arithmetic operators for the input IW, and is used to control the forwarding of data back to the ALU. This stage outputs the input IW to the Memory stage.

**Memory (MEM) Stage:** This stage decodes the DM instructions for the DM block, either loading/storing data from/to the DM itself, or simply forwarding ALU output results further along to the write back stage. This stage outputs the IW to the write back stage, or, upon detection that an I2C operation is complete, will stall the pipeline to issue this IW to the write back stage instead.

**Write Back (EX) Stage:** This stage records the program instructions in-order, in a Re-order Buffer (ROB). The ROB is a 10-entry, FIFO buffer that records each IW, it’s result, whether it’s complete, and whether it’s speculative. It reads the PM IW and buffers it, and reads the IW from the MEM stage to determine whether data can be committed back to the RF or whether the ROB must buffer this complete data.

Table 8 below shows how inter-CU stage stalls impact other stages.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Table 8: Stall Signal Table | | | | | |
| Stall Signal  Origin | Effect on Stage | | | | |
|  | LAB | ID | EX | MEM | WB |
| LAB\_stall | None | None | None | None | None |
| ID\_stall | - | - | - | - | - |
| EX\_stall | - | - | - | - | - |
| MEM\_stall | Stall | Stall | Stall | None | None |
| WB\_stall | Stall | None | None | None | None |

**LAB and Handling Jumps and Branches:** Jumps are the simplest flow control instruction. They are immediate and unconditional, and the Program Counter (PC) value is simply updated with the address in the jump IW.

Branches, if dependent on an instruction not in the pipeline or awaiting commit back to the RF, will simply execute and update the PC according to the branch address, which is issued subsequent to the branch IW itself. Otherwise, if the branch operands are still in the pipeline, the branch is buffered in the ROB, and all subsequent instructions buffered in the ROB are marked ’speculative’ until this branch is resolved. Additionally, the branch address is buffered in the branch\_addrs buffer.

Once the branch condition is known (results\_available = ‘1’) and the condition is not met (condition\_met = ‘0’), all entries in the ROB subsequent to this branch, and prior to the next branch (if one exists), will be marked as non-speculative (specul = ‘0’). Conversely, if the branch condition is known (results\_available = ‘1’) and the condition is met (condition\_met = ‘1’), all entries in the ROB for this and all subsequent instructions are flushed, and the PC is set to the current PC value plus one (PC\_reg + ‘1’). When condition\_met = ‘1’, all speculatively fetched instructions in the LAB and pipeline are flushed, and all ‘valid’ signals for those registers which were incorrectly, speculatively accessed are set high again. This represents a slight operational latency and presents an opportunity to improve the branch prediction algorithm, potentially by using a 1- or 2-bit prediction algorithm.

This CPU handles any number of branch statements issued from the PM. This is accomplished by tracking only the first two branches in the ROB. If the first branch in the ROB is resolved, the second branch in the ROB then becomes the first, and so on.

**Hazards:** This architecture was chosen to support a highly parallel paradigm whereby the digital inputs I/Os may be read, ALU results may be stored in DM, and RF contents may be issued to the ALU, all during the same clock cycle. However, there are still a few hazards that require special attention.

**Structural Hazards:** This architecture only poses two structural hazards: any Data Memory (DM) store must be allowed an additional clock cycle to complete if the subsequent instruction is a load from the same memory location, and instructions that rely on the result of an IO read result cannot utilize data forwarding. This was based on timing and complexity considerations, and the desire to prioritize implementing ALU data forwarding vice IO read data forwarding.

**Control Hazards:** This architecture poses the simple control hazard of not allowing the issuance of speculative DM stores, digital output writes, or I2C writes. While it is possible to incorporate these features and eliminate these hazards, from the pipeline issuance perspective, the complexity of the hardware required to buffer and execute the instructions when they become non-speculative did not outweigh the few clock cycles of speed up, given that branches are typically resolves in 0-5 clock cycles. Otherwise, this architecture always speculatively executes the non-branched instruction (i.e., Program Counter (PC) + 1). One implementation of a work around for this was explored as is discussed below.

**Analysis:** This CPU has successfully executed three different test programs, each testing a function of the CPU. These test scripts are attached as appendices to this paper. In the presented worst-case script, which repetitively wrote to the same register in an attempt to stress the data hazard detection functionality, the CPU completed this program with an average clock per instruction (CPi) of 2.33. In the presented best-case script, which issued more data-independent instructions, the CPU completed this program with an average CPi of 1.6. These numbers should be compared against a CPi value of 5, which it would be for a simple, non-pipelined processor, thus providing a maximum speedup of 212.5%. Note that an even more data-independent program written to maximize the number of correctly executed branches would operate quicker than a CPi of 1.6.

**Additional Considerations:**

A store buffer used to address the problem of memory disambiguation was explored, whereby speculative, out-of-order store instructions could be issued to the pipeline. This was intended to preclude overwriting valid data in the data memory with incorrectly, speculatively issued instructions. The store buffer (st\_buff) was a FIFO buffer that 1) buffers speculative store instruction, 2) clears the bottom-most instruction because it is not speculative, and 3) updates entries every clock cycle to determine whether they are still speculative, based on the state of the ROB. Additionally, all incoming store instructions were checked at the time of arrival at MEM\_top, not when the instruction is issued into the pipeline. This ensured that the latest speculative or non-speculative status is used when handling these instructions. This store buffer introduced an unreasonable amount of complexity into the design, given that speculative stores can just be stalled in the LAB, and was only partly functional. Therefore, the store buffer was removed from the design, which consequently also sped up synthesis from ~7 minutes to ~3 minutes.

During construction of this CPU, it was decided that the lower 16-bit result of multiplications should be written back to the destination register. This is because this case applies to most multiplications and would be the easiest to implement such that data was not lost. Therefore, care should be taken when using MUL and MULI instructions to avoid data overflow and data loss.

As assembler has been written to accommodate this ISA, Assembler.py. Simply place your assembly file in the same directory as the python assembler and it will output a correctly formatted MIF for your PM, or the error(s) that prevented successful assembly.

Appendix A: Loop Test Program

% first, initialize the loop index and a register %

00000000000 : 1100101010000000; % ANDI R21, 0x00 %

00000000001 : 1100101010001101; % ORI R21, 0x03 %

00000000010 : 1100000000000000; % ANDI R0, 0x00 %

00000000011 : 1100000000001101; % ORI R0, 0x03 %

% now on to the loop %

00000000100 : 0001101010000101; % loop: SUBI R21, 0x01 %

00000000101 : 1101011100000000; % CP R14, R0 %

00000000110 : 0001000000001001; % SUBI R0, 0x02 %

00000000111 : 0010000001010100; % MULT R0, R21 %

00000001000 : 1000011100000010; % ST R14, 0x0001(R0) %

00000001001 : 0000000000000001; % 0x01 %

00000001010 : 1000111110000000; % LD R31, 0x0001(R0) %

00000001011 : 0000000000000001; % 0x01 %

00000001100 : 0000000000000000; % ADD R0, R0 %

00000001101 : 1010101010000000; % BNEZ R21, loop %

00000001110 : 0000000000000100; % [loop address] %

% create some speculative instructions %

00000001111 : 1100000000000011; % NOT R0 %

00000010000 : 0101101011010100; % ROTL R21, R21 %

00000010001 : 1001000000100010; % done: JUMP done %

Appendix B: Data-Dependency Test Program

00000000000 : 1100101010000000; % ANDI R21, 0x00 %

00000000001 : 1100101010010101; % ORI R21, 0x05 %

00000000010 : 1100000000000000; % ANDI R0, 0x00 %

00000000011 : 1100000000001101; % ORI R0, 0x03 %

00000000100 : 0000000000000000; % ADD R0, R0 %

00000000101 : 0001000000010001; % SUBI R0, 0x04 %

00000000110 : 0010000000000000; % MULT R0, R0 %

00000000111 : 0111000001010101; % SFRA R0, R21 %

00000001000 : 1010000000000000; % BNEZ R0, n\_zero %

00000001001 : 0000000000010011; % [n\_zero address] %

00000001010 : 1100000000001101; % ORI R0, 0x03 %

00000001011 : 1011000000000001; % IOW R0 %

00000001100 : 1011000000000000; % IOR R0 %

00000001101 : 1010000000000001; % BNE R0, R0 %

00000001110 : 0000000000010011; % [n\_zero address] %

00000001111 : 0001000000010001; % SUBI R0, 0x04 %

00000010000 : 0010000000000000; % MULT R0, R0 %

00000010001 : 0111000001010101; % SFRA R0, R21 %

00000010010 : 1001000000100100; % zero: JUMP zero %

00000010011 : 1100000000001101; % n\_zero: ORI R0, 0x03 %

00000010100 : 1001000000101000; % done: JUMP done %