A Simple MIPS Pipelined processor Implementation

Author: Nour Nawar
Date: 1/8/2025

Preface

This document contains detailed information on two simple implementation schemes of the MIPS core instructions. The motive behind such a project was to understand the philosophy behind building hardware that can understand software-written instructions. Getting a feel for such a layer enables understanding and writing better software. This project implemented various fundamental hardware units and showcased how they connect with each other.

This document is intended for students and hobbyists interested in the low-level design and implementation of a basic RISC processor. It assumes basic experience with logic design and computer organization. For that, it is recommended to read at least chapter 1 and Appendix B from the amazing book “Computer Organization and Design: The Hardware/Software Interface.”

Table of Contents

Introduction

MIPS stands for Microprocessor without Interlocked Pipeline Stages. It is one of the oldest RISC instruction sets and also one of the simplest. It shares a lot with modern RISC instruction sets like ARMv7 and ARMv8[1]. What makes RISC-based instruction sets of architecture (ISA) stand out from their relative CISC-based counterparts is their profound simplicity. This attribute makes understanding such instruction sets of architecture fairly easy and simple to implement. Furthermore, a simple ISA is easy to predict and has fewer exceptions. **Simplicity favors regularity.** Furthermore, this key idea makes implementing a pipelined version easier.

After understanding the MIPS core instructions and how they work on the software level, implementing a simple single-cycle scheme is in order. Such an implementation contains three out of the **five classic components of a computer**, which are an ALU, a control unit, and a datapath[2]. At the point such a scheme is realized, it is conceived that it is unrealistic due to its inefficiency. From there the idea of pipelining stems and a more practical implementation is studied. Pipelining exploits the idea that multiple instructions can execute synchronously. This allows for a higher throughput for a large number of instructions.

1 Wikipedia page https://en.wikipedia.org/wiki/MIPS_architecture
2 Memory units used in the implementations are external to the CPU and are abstracted.

Overview of MIPS Core Instructions

Instructions are the language of the computer. It is the language that hardware can understand. The vocabulary that composes these instructions is called the instruction set of architecture. The great idea of stored computer programs means that these instructions can be packed to form the useful program, and then this program can reside inside of a certain instruction memory and be executed accordingly. The MIPS core ISA is composed of three classes of instructions: arithmetic, memory, and branch. Arithmetic instructions operate on processor registers, and results are asserted into processor registers. Memory instructions must access memory either to write or to read. Such a class must also have access to registers. Finally, the branch can be broken into two types: conditional branches and unconditional branches. Unconditional branches change the program counter content to a specified address. This operation is unconditional. Conditional branches change the PC content based on a register-register comparison.

A diverse set of instructions lie under these three categories and possibly others. However, the implementation is concerned with a subset of MIPS core instructions. These instructions are limited to integer operations and do not support floating-point operations. Out of these core instructions, only the following are implemented: **"add, sub, and, or, slt, lw, sw, j,** and **beq."**

The arithmetic instructions (**add**, **sub**, **and, or,** and **slt**) are fairly straightforward. They all take the following format: "instruction $reg1, $reg2, $reg3," where the operation is performed on reg2 and reg3 and the result is stored in reg1. The set on less than (**slt**) sets a register data content to a one if the two other registers are equal. For example, "slt $reg1, $reg2, $reg3" sets reg1 to one if reg2 and reg3 are equal. Here, reg1 is called the destination register (rd), reg2 is called the first source register (rs), and reg3 is called the second source register (rt). The following figure shows the machine code[3] decoding of this class of instructions.

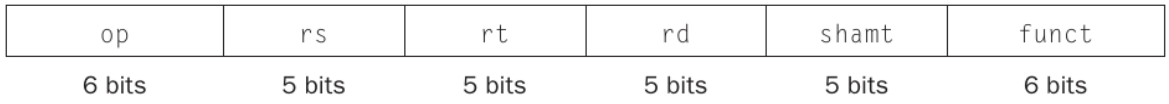

Figure 1: R-type instruction format

The load word and store word (**lw** and **sw**) memory instructions use the same format, that is, "instruction $reg1, offset($reg2)." For a store, the data content of reg1 is stored inside of memory location "offset + data content of reg2." Note that this memory location must be word aligned; that is, it must be the start address of a word, which is four bytes. Below is the machine code format for this class of instructions.

3 A machine code for an instruction is the binary representation of such instruction understood by the computer and stored in the instruction memory.

Figure 2: I-type instruction format

Branch instructions are a bit more tricky because not all of them have the same format. For instance, the "beq" instruction has the same format as the memory instructions shown above. However, the address that is found in the address field is not used directly. Instead, this instruction uses **pseudo-direct addressing mode**, where the actual branch address is the constant field shifted to the left by two and then added to the program counter. The unconditional jump instruction "j" uses a different instruction format given below. It also uses a different addressing mode called **PC-relative addressing**. In this mode, the address is formed by shifting the 16-bit constant field to the left by two and then concatenating it with the upper two bits of the PC.

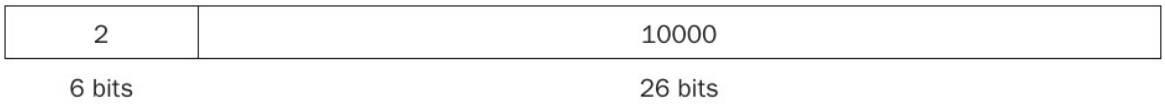

Figure 3: J-type instruction format

Representing instructions as numbers is a great idea that facilitates how computers understand what programmers write. Using memory to store these numbers so that the processor can execute the program as a whole is another key idea.

An Implementation Approach

A processor that is able to read a program in the form of instructions and then execute whatever that program does must logically follow through some predefined steps.

1. It must first fetch the instructions from the instruction memory.
2. It then decodes that instruction to know its type, what controls are needed, what registers are used, and whether immediate fields from the instruction are used.
3. It goes through the Arithmetic Logic Unit (ALU) to either do an arithmetic operation or to calculate a branch address.
4. Then the result gets forwarded into the data memory, where it addresses a specific address for reading or writing in the case of a memory instruction, or it can pass through if it is an arithmetic instruction.
5. Then the result from the memory or the ALU gets written back to the register file, where all the registers are found.

Fetching an instruction is easy; that specific instruction is loaded from the instruction memory. Decoding an instruction involves resolving which registers are used as the source, second source, or destination. It also involves generating the right control signals for the instructions. The ALU must be able to perform the right operation based on the instruction inputs it receives; for that, another control unit for the ALU is required. Such a control unit takes as input the op-code from the instruction and a control signal from the main control unit. Next is the memory access. The control unit determines whether a read or a write is in order or to bypass it completely, which is the case for an arithmetic instruction. Writing back is also controlled by the control unit, which chooses which value is to be written back to the register file. This value can come from the memory or from the ALU.

Simulation Software

Logisim[4] is a free and open-source software that is used to simulate this project. A simulation tool like this abstracts details about how things are put together and how timing works, but it was a nice trade-off for simplicity. The following sections all capture components and datapath from inside the simulation. It is recommended that you install the program and open the simulation file yourself so that you follow along with what is being said, although the photos and documentation throughput this document should be enough to grasp the big idea.

Logisim offers a library of ready-to-use gates, components, and even TTL IC chips; these won't be used. For example, memory elements are supported as components that can be used out of the box. This abstraction makes the design easier. However, there are a few downsides to a tool like this. First, timing analysis is not perfect, and it is not easy to tweak clock signals like you would easily do with an HDL. Second, memory elements are not byte-addressable and cannot be made so, unless you design them this way, which is not easy.

Apart from that, the whole design is going into two **.circ** files: one for the single-cycle implementation and another for the pipeline implementation. These files include the sub-circuits for all the components as well.

4 https://github.com/logisim-evolution/logisim-evolution

MIPS Single Cycle Implementation

Introduction

A single-cycle implementation scheme is one in which each instruction takes exactly one clock cycle to execute. For such a scheme, every component of the processor is wired directly to the next component. The five main phases of the processor are just wired with the required multiplexers and control wires. This section covers such an implementation in detail.

The PC

The program counter is simply a memory register that stores the address of the next instruction to be executed. The program counter should increment by one word, or four bytes, every cycle in case of a non-branch instruction. In case of a branch instruction, the branch address should be calculated according to the branch type and then fed back to the PC. In order to choose between which of these two instructions is fetched into the PC, a multiplexer is used.

The PC is preceded by two multiplexers. The first one chooses between whether to give the PC the next sequential instruction or to branch to an address given by a beq instruction using pseudo-direct addressing. The second multiplexer chooses between proceeding to one of these previous addresses or to branch to an address given by a jump instruction. The following image shows the PC datapath.

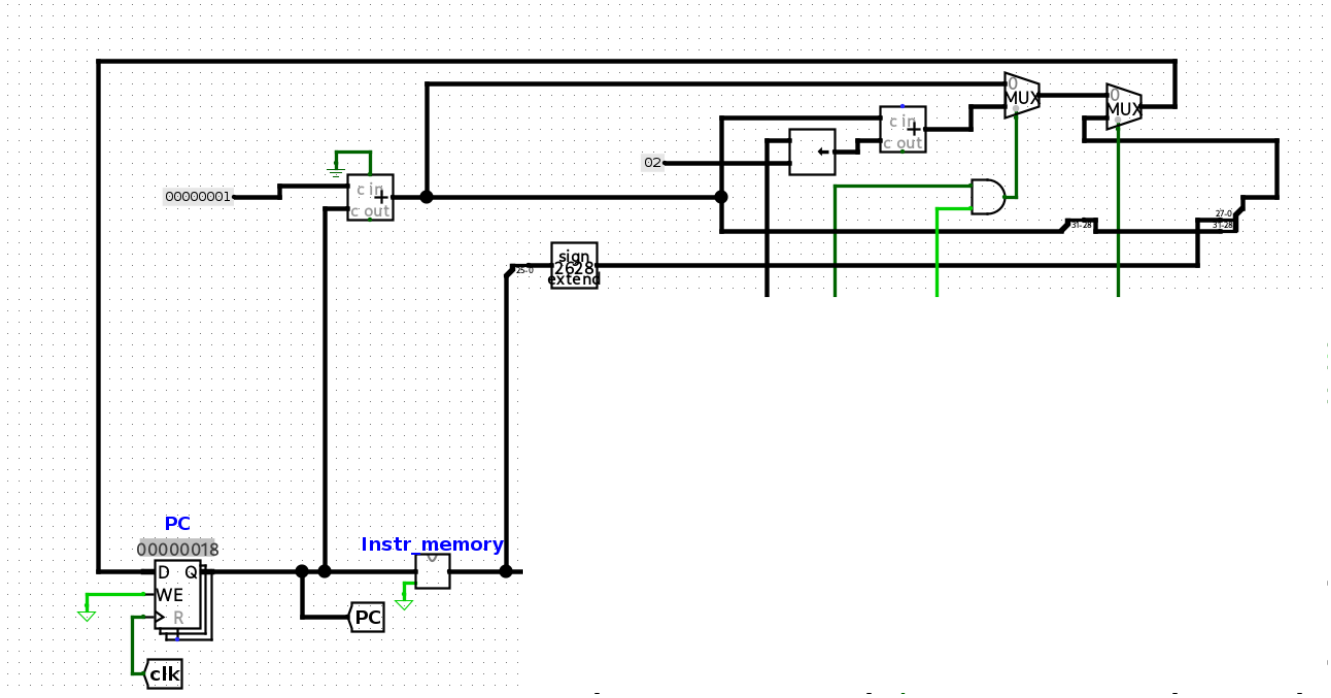

Figure 4: The datapath for the program counter

Notice here that the PC is added with the constant 1 before it is operated on. This is because Logisim uses word alignment (each address is a word not a byte). The two inputs to the first multiplexer are the PC + 1 and a sign extended 32-bit signal fed from the instruction [15-0]. In case of no jump instruction, the output from the first multiplexer passes back to the PC. If there is an unconditional jump, the **jump** control signal from the control unit chooses the other input to the second multiplexer. This second input is the instruction fields [26-0] shifted left by two and then concatenated with the upper four bits from the PC. A shift-left component and a sign-extent components are used. As for the beq instruction, it is triggered when the **branch** control signal is asserted and the zero output from the ALU is asserted. The ALU's zero output is asserted when the two input registers have the same data content.

The Instruction Memory

The instruction memory is a read-write memory in which the program is loaded. It has a single input, which is the address, and one output, which is the instruction. The PC supplies the address of the instruction, and the output follows into the decoding stage. The following figure shows the instruction memory sub-circuit.

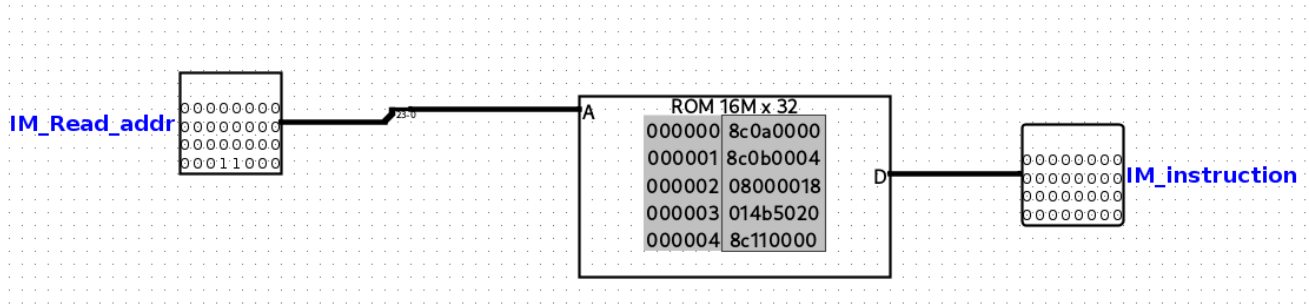

Figure 5: The instruction memory.

The Control Unit

The control unit produces the control signals for the datapath. It takes as input the instruction's opcode and outputs 10 different control signals. These signals are as follows: RegDst, Branch, MemRead, MemtoReg, ALUop [1-0], MemWrite, ALUSrc, RegWrite, and Jump. The following table specifies the use of each of these control signals.

Table 1: The 10 control signals used in the processor.

Control signal	Use
RegDst	Selects the destination register (rd or rt). Asserted → chooses rd Deasserted → chooses rt
Branch	Selects the branch address associated with the beq instruction
MemRead	Enables reading from the data memory
MemtoReg	The write-back to the register file comes from the data memory
ALUop	Two bits used to control the operation of the ALU

MemWrite	Enables writing to the memory
ALUSrc	Asserted → second source of the ALU comes from the instruction immediate field Deasserted → second source comes from the register file
RegWrite	Enables writing to the register file
Jump	Selects the branch address of the jump instruction

As an example, it would be expected that an add instruction should have the following control signals asserted: RegDst, RegWrite, and a specific ALUop for the ALU operation of addition. For an lw, the following should be asserted: MemRead, MemtoReg, ALUSrc, RegWrite, and a specific ALUop for the ALU operation of addition. The following table showcases all possible control signals for all instructions.

Table 2: Control signals for each instruction. Control bits ordered as follows: [RegDst..Branch..MemRead..MemtoReg..ALUop1..ALUop0..MemWrite..ALUSrc..RegWrite..jump] Notice that all of the arithmetic instructions share same control signals. What controls the operations performed by the ALU is the ALU control unit, which takes input the ALUop from the control unit and the funct field from the instruction. More about that in the next section.

Instruction	Control
add	1000100010
sub	1000100010
slt	1000100010
and	1000100010
or	1000100010
lw	0011000110
sw	0000001100
beq	0100000000
j	0000000001
nop	0000000000

The following figure shows the logic-level implementation of the Control Unit.

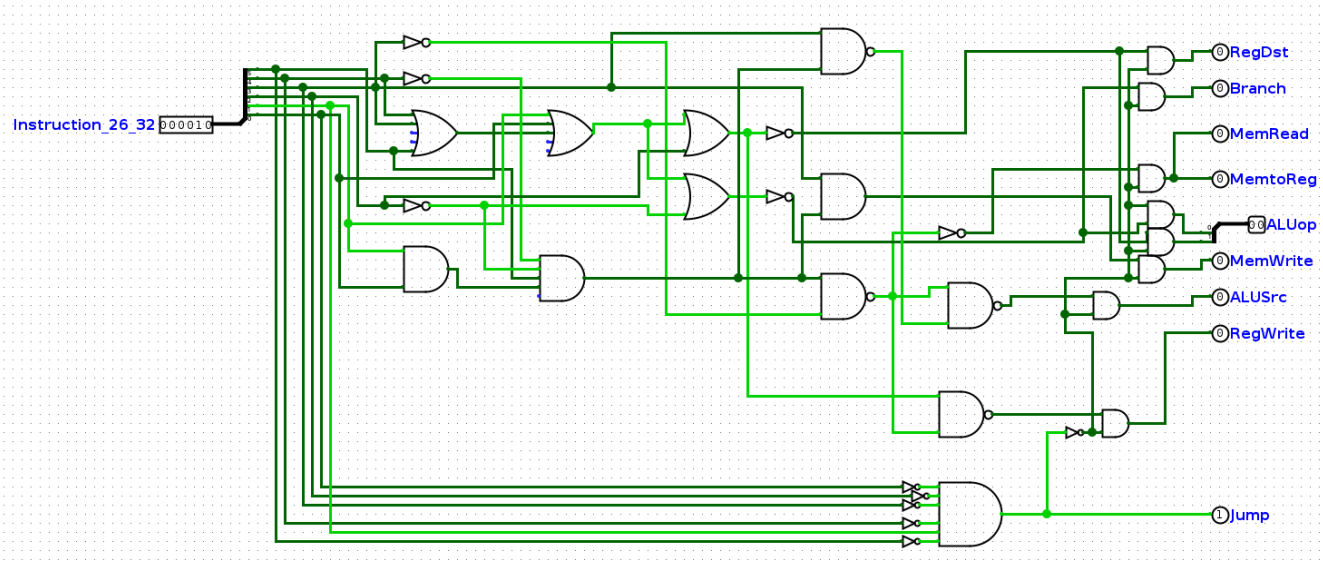

Figure 6: The Control Unit

The Register File

The register file is where all registers exist. It has five inputs and two outputs. The first two inputs specify the two addressed registers. There is a reg1 input and a corresponding reg1 output, and the same holds for reg2. The write register input specifies which register is to be written to. The write register data holds the data to be written. The RegWrite control signal controls the write process; it either enables it if the instruction is intended to write to a register (like **add** or **lw**,) or inhibits it for other instructions (such as **beq** and **sw**.)

Individual registers have three inputs that are of concern: data, clock, and enable. The data input is 32 bits. The enable specifies whether the register stores at the rising edge of the clock. There are two possible operations that can be done on a register: reads and writes. To read a register, you must select it using the 5-bit input to either register1 or register2. This select input goes into a 32-bit multiplexer that chooses which register is output on the register1 bus and register2 bus. To write to a register, the write enable control bit should be asserted. Also, the number of that register is used as a select input to a 32-bit decoder. The enable input of each register is connected to the decode's output ANDed with the write enable signal. When the clock rises, the value on the write

register input gets stored in the selected register. The following figure shows the internal sub-circuit for the register file.

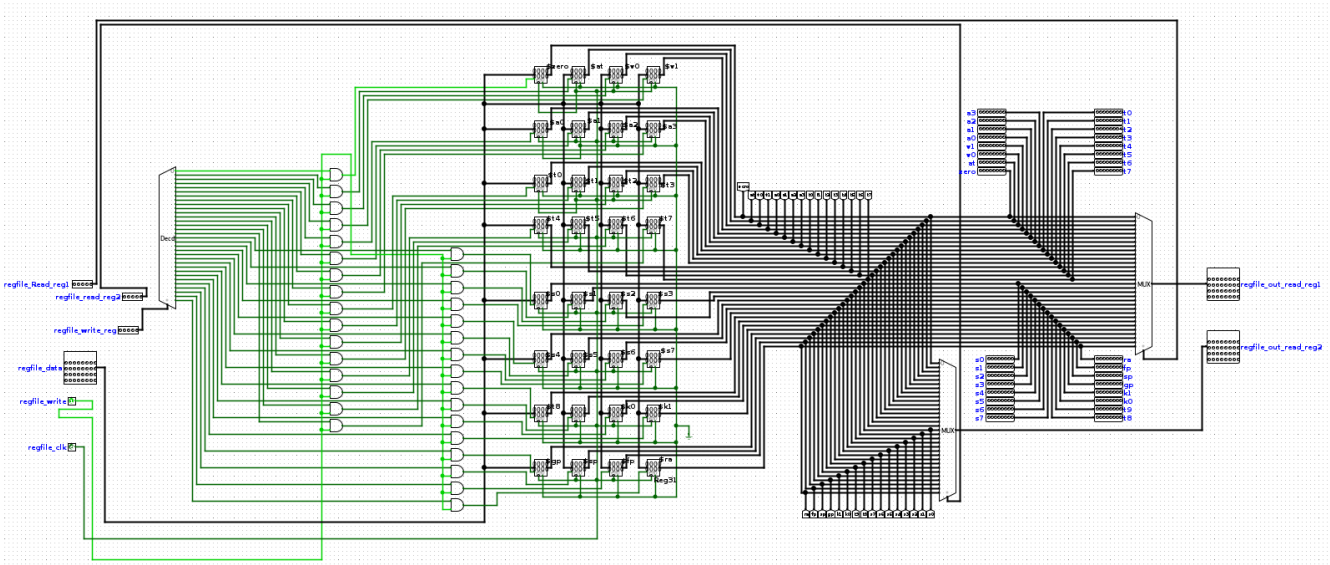

Figure 7: The register file

The registers are put in chronological order as with the classic MIPS architecture. Figure 8 shows the numbering standard for the MIPS-32 registers along with the convention for using them.

name	reg#	convention
$zero	0	**constant** 0
$at	1	reserved for compiler
$v0-$v1	2–3	**results**
$a0-$a3	4–7	**arguments**
$t0-$t7	8–15	(callee-saved) temps
$s0-$s7	16–23	**caller-saved**
$t8-$t9	24–25	(callee-saved) temps
$k0-$k1	26–27	reserved for OS
$gp	28	global pointer
$sp	29	**stack pointer**
$fp	30	**frame pointer**
$ra	31	**return address**

Figure 8: The register naming convention for the MIPS-32 instruction set

The ALU

The arithmetic and logic unit is responsible for all calculations needed for instruction execution. It has seven inputs and four outputs in the chosen design. The inputs are as follows.

1. Input A, which is the first input operand.
2. Input B, which is the second input operand.
3. Carry in, for future expansions.
4. ALU A_invert, which is used to invert the bits of A.
5. ALU B_invert, which is used to invert the bits of A.
6. ALU control unit output, which is fed to the ALU and determines the operation done by the ALU.

The ALU is 32-bits and is made up of eight 4-bit carry-lookahead ALU subunits. Each of these four bit adders contain four 1-bit ALU units. Figure 9 shows the 1-bit ALU circuit.

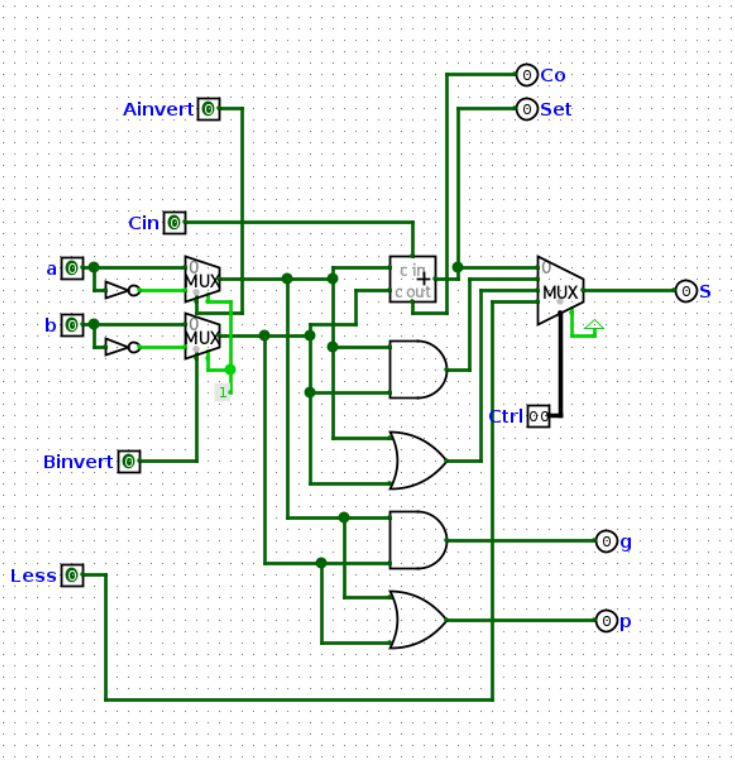

Figure 9: 1-bit ALU

The 32-bit final ALU is an expanded version of the 1-bit ALU. It uses the same inputs and the same controls. For the 1-bit inputs “a” and “b”, the Ainvert and Binvert signals control the multiplexer, which chooses between the input and its inverted version. The Cin is the carry input from other ALU subunits. The Less input is used for the set on less than “slt” operation. Finally, the 2-bit Ctrl signal chooses which of these inputs to choose. The following truth table shows that.

Table 3: ALU control signals and their respective operations

Ctrl	Operation
00	Add and subtract
01	Logical AND
10	Logical OR
11	Set on less than (slt)

Using this 1-bit ALU, we can build a 4-bit ALU by combining four of these 1-bit ALUs. Figure 9 shows the 4-bit ALU. Notice that NORing individual data bits is used to check equality. The output for such equality is ALU4_eq. This can be extended to the 32-bit ALU by simply ANDing individual ALU4_eq outputs. The ALU4_setout is the output from the MSB of the 4-bit ALU. This value is used in the slt logic.

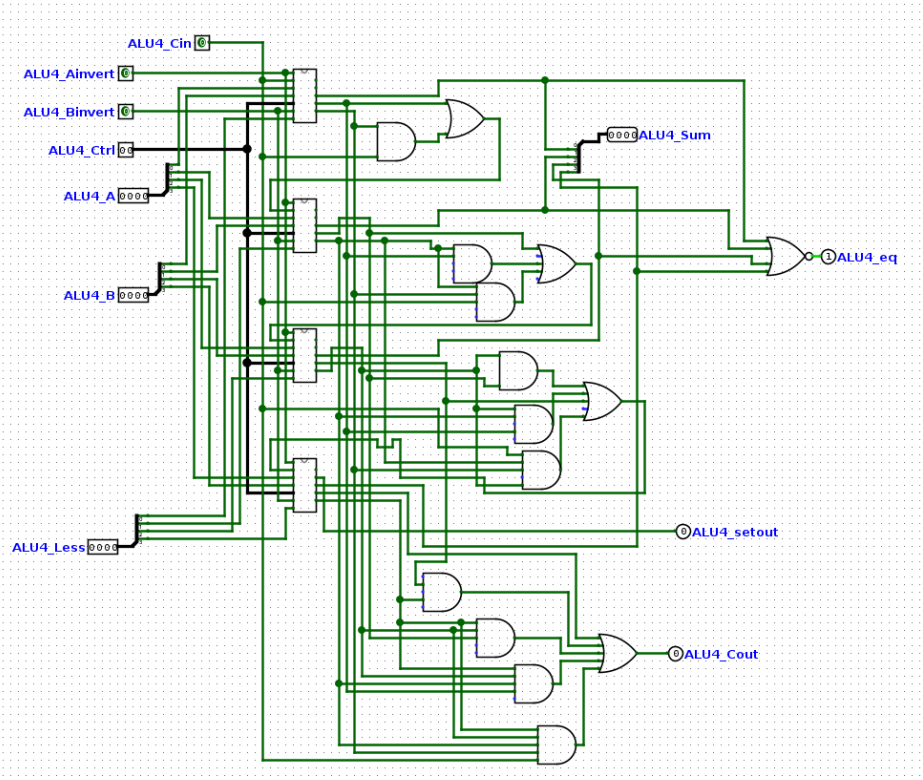

Figure 10: 4-bit ALU with carry-lookahead addition

The set on less than instruction stores a one in the destination register if the first source register is less than the second source register. To check for this, we can simply subtract rt from rs and if the MSB is a one, then A is less than B. We can forward this MSB output to the LSB while setting all other bits to zeros. To do this, B_invert and Cin are asserted. The ALU4_setout of the last 1-bit ALU in the last 4-bit ALU is used in the final 32-bit ALU. This result is extended to 4-bit width and then forwarded back to the Less input of the LSB. Figure 10 shows the final 32-bit ALU.

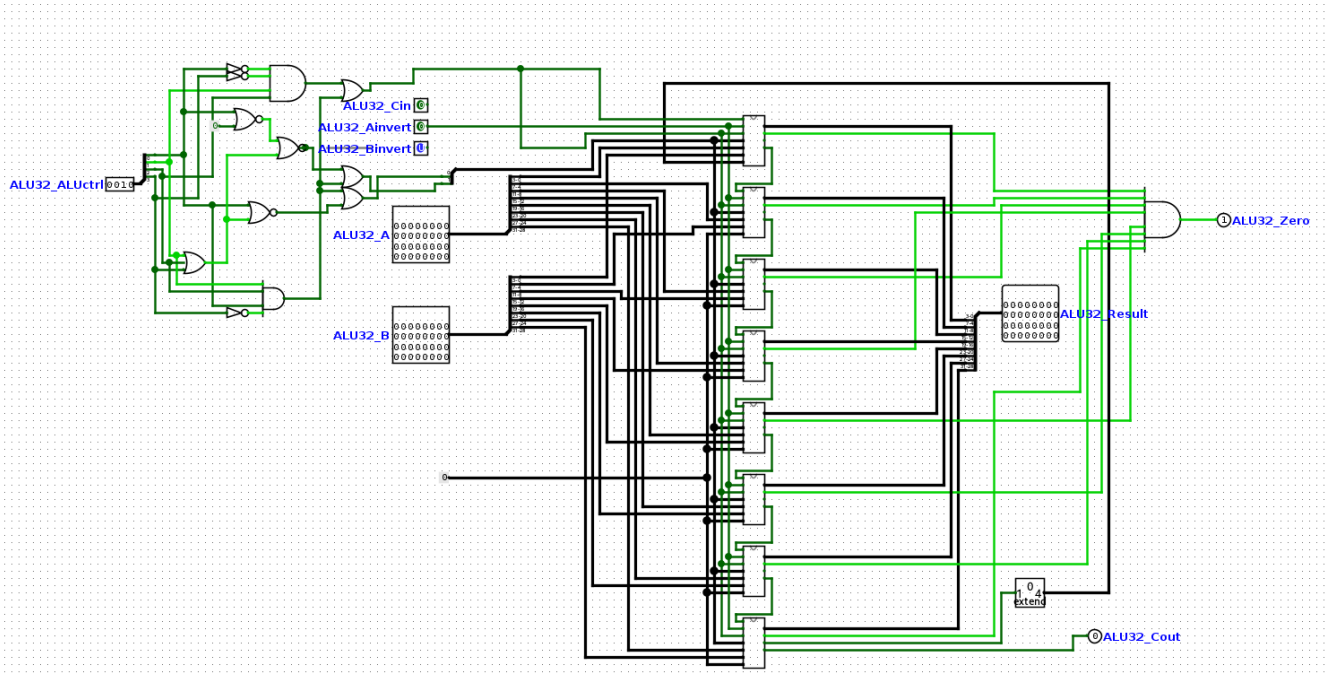

Figure 11: 32-bit ALU

The ALU control unit decides which operation is performed. It takes input the ALUop control signal from the main Control Unit and the funct fields [5-0] from the instruction. Table 4 shows the ALU Control unit logic.

Table 4: The ALU control output for each instruction. Note that the Funct fields and the ALUop are inputs. For the sub instruction, Binvert and Cin must be asserted. This is done via a simple external logic.

Instruction	ALUop	Funct	Operation	ALU control
lw	00	xxxxxx	add	0010
sw	00	xxxxxx	add	0010
beq	01	xxxxxx	sub	0110
add	10	100000	add	0010
sub	10	100010	sub	0110
and	10	100100	AND	0000
or	10	100101	OR	0001
slt	10	101010	slt	0111

The Data Memory

The data memory is readable and writable. It has five inputs: address, data, write enable, read enable, and a clock signal. The write and read enable signals are controlled by the control unit. If the instruction is a **sw**, the write enable is asserted. If the instruction is a **lw**, the read enable is asserted. Memory reads are asynchronous; this causes no problem because writes to the register file are clocked.

The address to be addressed comes from the ALU; this is because the address in the instruction's source register is added with an offset. The data written to the memory comes from the second source register, rt, directly. Following the data memory is a multiplexer that chooses whether the write-back data comes from the ALU, like in the case of an arithmetic instruction, or from the data memory, like in the case of a store instruction. Figure 11 shows the datapath of the data memory.

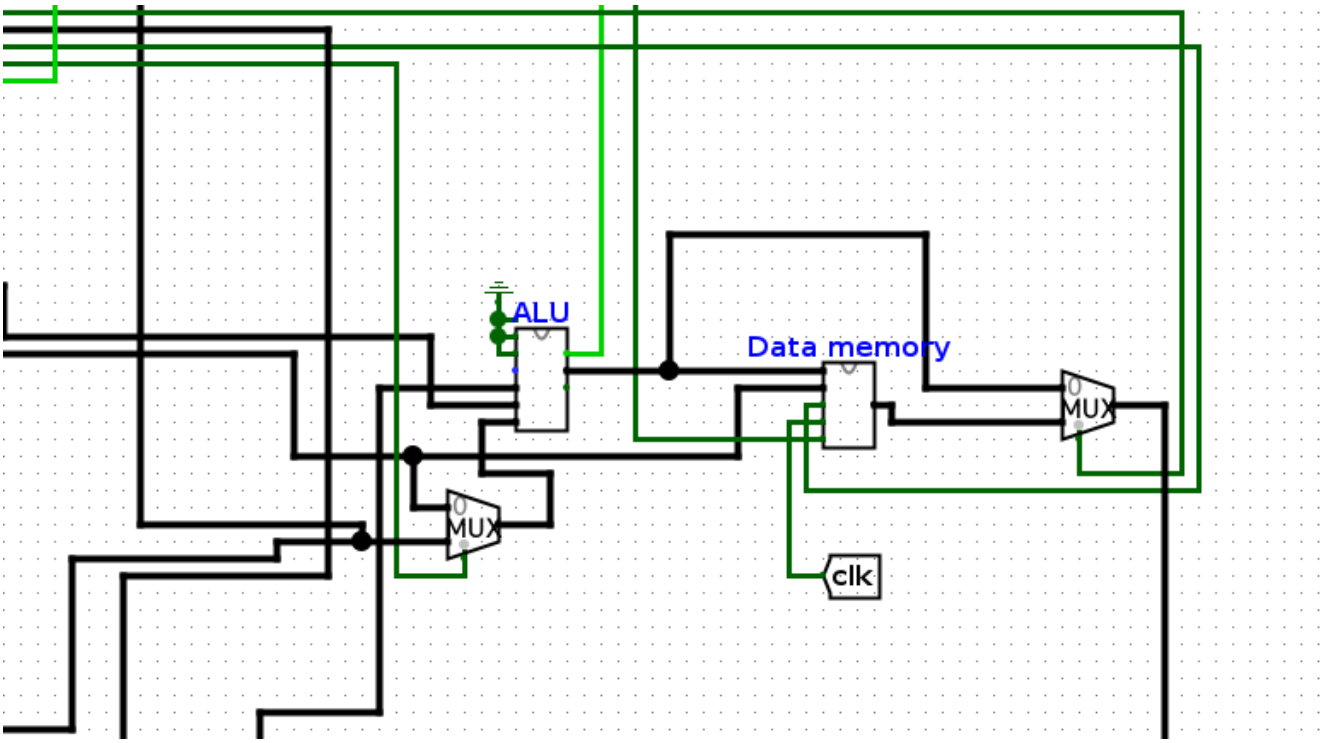

Figure 12: The datapath connecting the ALU and the data memory. Notice that the MemtoReg control signal coming from the ALU decides whether the output of the ALU or the output from the data memory is written back to the register file.

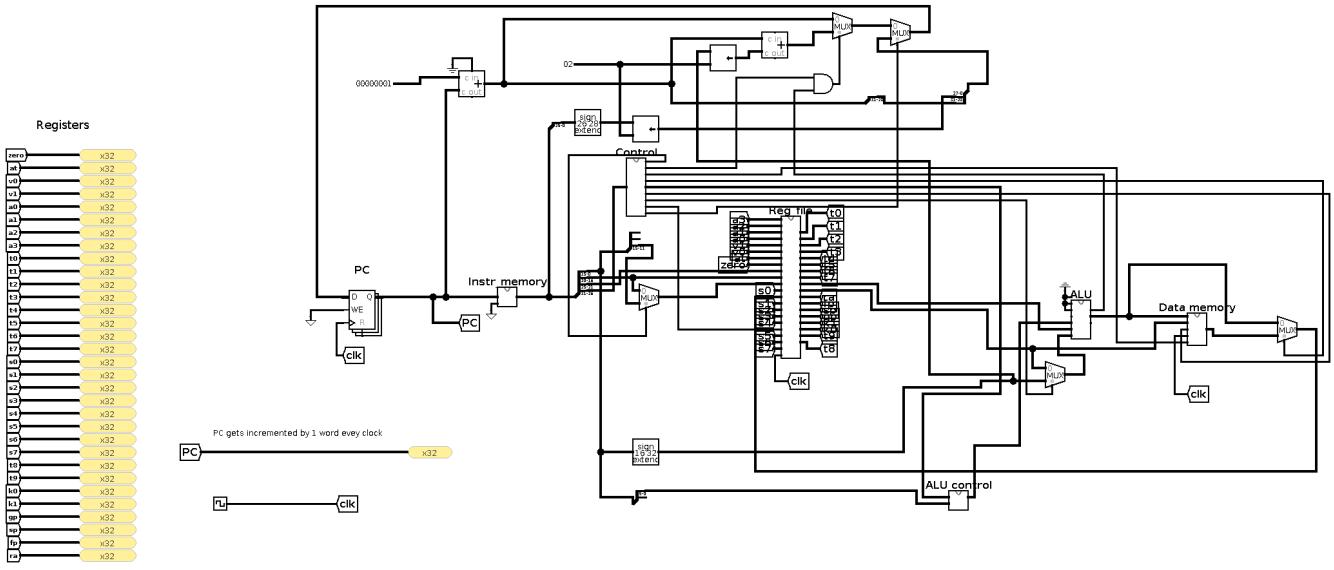

Figure 13: Full Datapath for the Single Cycle Implementation

MIPS pipelined processor

Introduction to Pipelining

The single-cycle implementation is an impractical implementation scheme. This is because of its low inefficiency. The single-cycle implementation takes one cycle for each instruction. This requires the clock cycle to be long enough to account for the longest instruction, which is lw in this case. This leaves instructions like the jump instructions with plenty of wasted time. And as the program gets bigger, the time inefficiency gets higher. An implementation scheme that solves this issue is pipelining.

Pipelining is a scheme in which the datapath is separated into different stages. Each of these stages can hold a separate part of a different instruction. This makes sure that the largest clock cycle needs to be as long as the slowest stage. This drastically fastens things up. The implementation scheme used has five stages: instruction fetch, instruction decode, instruction execute, memory access, and write back. These five stages may be abbreviated as IF, ID, EX, MEM, and WB. In every pipeline stage, a part of an instruction is being progressed. For example, an add might be decoded while a load is calculating an address in the execute stage. Because every instruction is independent from all other instructions[5], each requires different control signals and datapath configurations. For such reason, pipeline registers are introduced. They act as temporary storage registers that contain all signals a certain instruction would need during a specific stage. Only four pipeline registers are needed for the five-stage implementation, and they are named as follows: ID/IF, IF/EX, EX/MEM, and MEM/WB. Each pipeline register is named according to the two stages it stands between.

5 Sometimes dependencies arise between different instructions that address the same registers. These dependencies cause hazards. More about hazards later in this section.

The Pipeline Registers

The IF/ID register stores the current instruction and the PC + 1. It also has a write enable input, which controls whether the PC increments or not, and a flush input to clear the instruction stored inside. The last two inputs are useful in adding stalls and flushing unwanted instructions. Synchronized by the clock, it outputs the PC + 1 and the instruction machine code. The ID/EX register stores the reg1 and reg2 output from the register file, the immediate part of the instruction [15-0], the control signals, and the source, second source, and destination register numbers. The last inputs are passed for use in the hazard detection unit found in the EX stage. The ID/EX register outputs the read reg1 and reg2 data content, the control signals, and the three register numbers. Notice that only a subset of the 10 instructions are used in the EX stage. These are ALUSrc, ALUop, and RegDst. The remaining instructions are passed down to the next pipeline stage. The EX/MEM register stores the output of the ALU, the reg2 from the ID/EX register, the destination register (which can be rt or rd), and the control signals. Also note that only MemWrite and MemRead are used in the MEM stage and that the remaining two control signals get forwarded to the MEM/WB register. Finally, the MEM/WB register stores the output of the ALU stored in the EX/MEM register, the output from the data memory, and the destination register.

Hazards and the Pipeline Datapath

The same components from the single-cycle implementation are used here, the control signals are unchanged, and the datapath is upgraded so it resolves hazards. There are two types of hazards that need to be addressed. The first are data hazards. Data hazards happen when an instruction first in sequence, later in the pipeline, writes to a register that is operated on by a later instruction, earlier in the pipeline. These hazards can typically be solved using a forwarding unit, which forwards the result directly from the other stage without waiting for it to be written back. The main forwarding unit is located in the EX stage and forwards results from the MEM and WB stages. There is also another forwarding unit for the branch instruction, which will be covered later. The EX forwarding unit can forward from the MEM stage or from the WB stage. For the MEM stage, it uses the data stored in the EX/MEM register, and for the WB stage, it uses data stored in the MEM/WB register. The name of the pipeline register is used, followed by a dot “.” and then the name of the output. The following logic is used to forward from the MEM stage.

if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)

and (EX/MEM.RegisterRd = ID/EX.RegisterRs))

That is, the register write signal for the earlier instruction is asserted (meaning the instruction writes to the register file), the destination register file is not zero, (because the zero register cannot be written to), and the destination register for the earlier instruction (EX/MEM.RegisterRd) is the same as the later instruction's first source file (ID/EX.RegisterRs). If this logic is true, the data from the EX/MEM register must be forwarded to the first source register of the first input of the ALU.

To forward from the WB stage, a similar logic holds but only differs in the destination registers; it becomes MEM/WB.RegisterRd. Also, there is a priority check; if both EX/MEM.RegisterRd and MEM/WB.RegisterRd equal Rs, EX/MEM.RegisterRd is used. The following shows this logic.

if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)

and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)

and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRs))

and (MEM/WB.RegisterRd = ID/EX.RegisterRs))

If this condition is satisfied, the register data is forwarded from the output of the multiplexer following the WB stage. All the logic statements above hold equally for the second source register, Rs. Thus, there must be two multiplexers, each choosing which data to forward according to the select signals coming from the Forwarding Unit. One multiplexer is used for the first operand to the ALU and the second for the other operand. The outputs from the Forwarding Unit are ForwardA and ForwardB. Forward A is the control signal for the first source register, and ForwardB is the source for the second source register. The following table shows possible values for the Forward output.

Table 5: Forward control signal output. Note that ForwardA and ForwardB work the same but for Rs and Rt respectively.

Forward	Selected input
00	The Rs or Rt from the ID/EX register
01	Forwarded value from the MEM/WB register
10	Forwarded value from the EX/MEM register

The following figure shows the Forwarding unit sub-circuit.

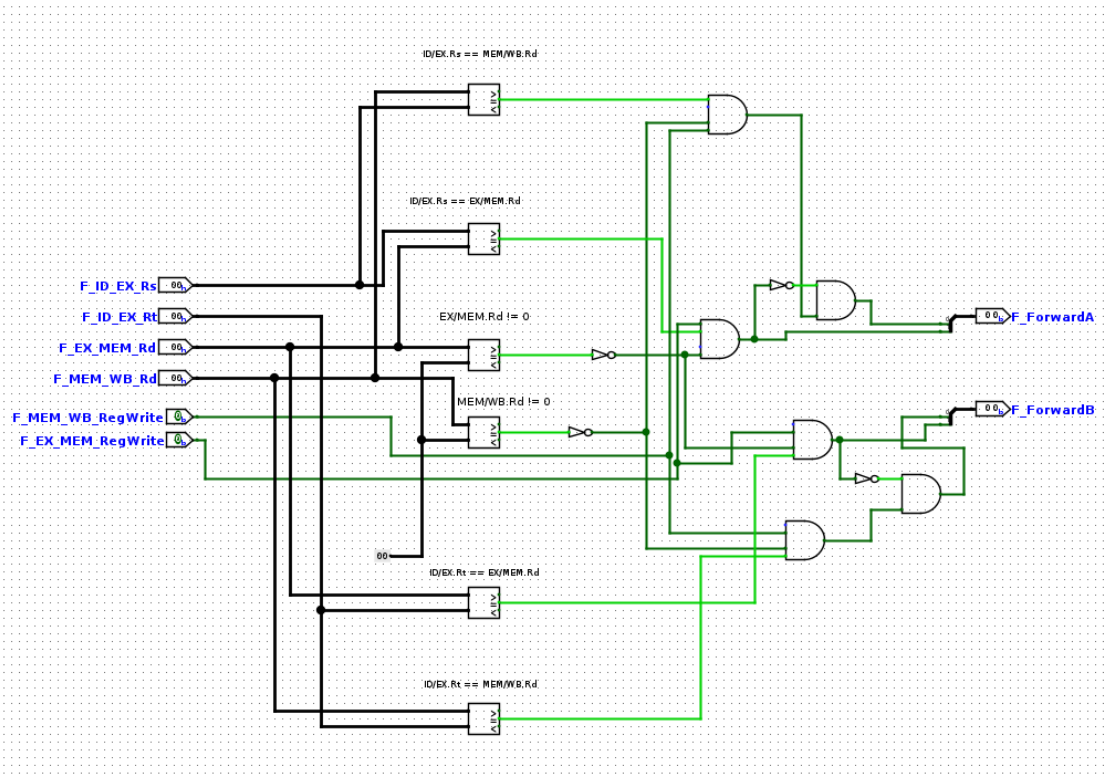

Figure 14: The EX-Forwarding Unit

Conditional branch instructions (i.e., beq) require the ALU for register comparison. Because branch instructions determine which instructions follow in order, they are better placed in an earlier stage so that decisions based on them can be made fast. Instead of a full-blown ALU, a simple comparator can be used for comparing the

two register operands. The address calculation can be used as before. There are some complications to this approach.

1. A dedicated branch forwarding unit is needed to resolve data hazards.
2. Sometimes dependencies can't be solved directly using the forwarding unit and stalls are needed.
3. The pipeline does not stop, so if the branch is taken, the next instructions must be flushed to avoid executing it down the pipeline. This is called branch-not-taken prediction[6]

Luckily, building such a forwarding unit is not difficult. Also, extra logic is added to the Hazard detection unit to add a bubble, or a stall, if the data dependency cannot be resolved right away by the Branch Forwarding unit. More sophisticated dynamic branching state machines can be used to reduce the time penalty. The simple static scheme is used in this implementation, though.

The Branch Forwarding unit uses the same inputs and outputs as the forwarding unit from the EX stage. The multiplexers, however, output directly to a comparator that yields the comparison result, which is used to select the branch address as the input to the PC. Note that the Branch Forwarding unit can technically forward results from the EX stage, but such a scheme would be complex and potentially unstable. **Good design demands good compromises**. It is easier to replicate the EX forwarding unit and add a simple logic to stall the pipeline when necessary.

The load instruction does not write its result until the last stage. Also, its result is not available for forwarding until the MEM/WB stage, because the instruction accesses memory. Due to that, data dependencies that require stalls are in order. The Hazard detection unit is used to insert bubbles whenever the lw instruction causes a load-data hazard. The following is the logic used for this component.

```
if (ID/EX.MemRead and ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
(ID/EX.RegisterRt = IF/ID.RegisterRt)))
        stall the pipeline
```

6 It is a type of static prediction where the next instruction is fetched regardless of the branch condition. If the branch is not taken, the instruction follows and we save time. Otherwise, the next instruction must be flushed and the PC must be changed to the calculated branch address.

The ID/EX.MemRead is only asserted for a load instruction. If the ID/EX.RegisterRt (which is the destination register for the load instruction) equals the IF/ID.RegisterRs, or IF/ID.RegisterRt, then the pipeline must be stalled. Such a stall is necessary to be able to forward the result from the WB stage to the ALU for operations.

In order to stall the pipeline, the PC must stop advancing, the control signals must not proceed down the pipeline, and the IF/ID register must stop writing. To stop the PC and the IF/ID register from writing, the write enable is deasserted. To stop the control lines from advancing, a multiplexer is added and a selects zeros as input when the Hazard detection unit asserts a control enable signal. When a lw instruction is followed by a beq, the pipeline must stall two cycles. This is because the lw cannot forward until the WB stage. Figure 15 shows the logic for such circuit.

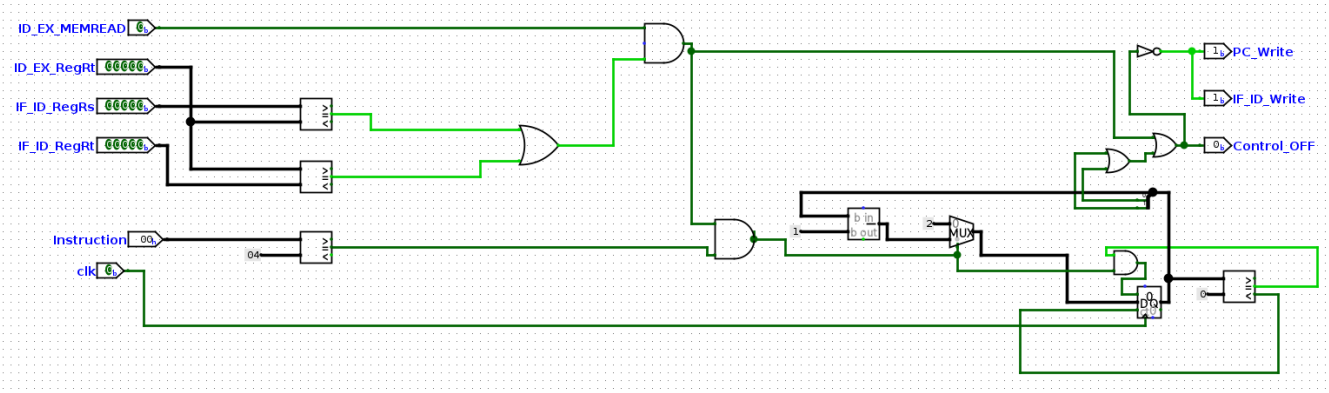

Figure 15: The Hazard detection unit. The upper AND gate checks for simple load-data hazards that require a single stall. The lower AND gate checks for the same logic and checks whether the instruction is a beq, which has an opcode of 0x04. If so, the simple counter state machine stalls the pipeline for two cycles.

L

The Hazard detection unit inserts all needed stalls. The Branch Forwarding Unit can thus function the same as the EX Forwarding Unit.

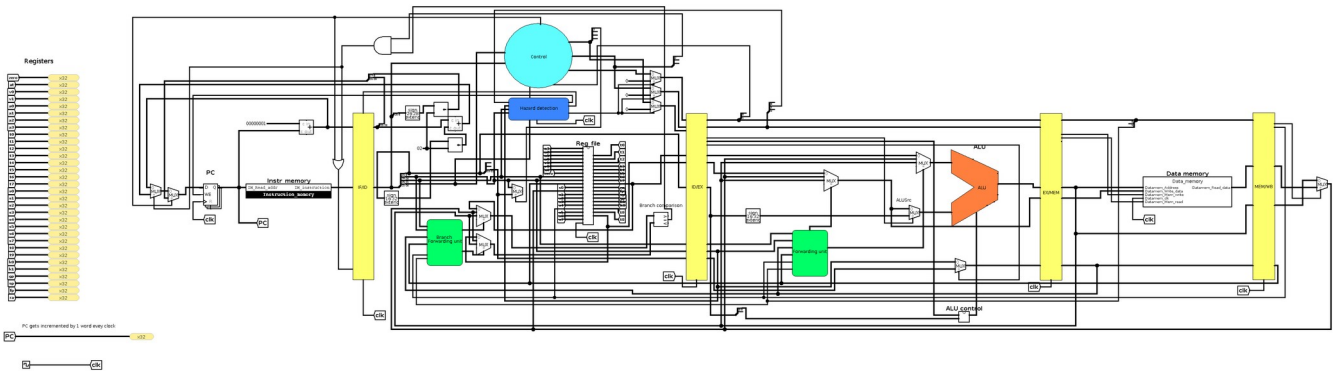

Figure 16: The Full Pipeline Datapath

Possible Improvements

Listed below are possible improvements to this project.

1. Explore using more pipeline stages. This approach can increase the clock frequency and throughput, but would introduce more dependencies and hazards.
2. Support more instructions from the core subset (e.g., addi, subi, addiu, subiu, jal, etc.). In a more advanced implementation, floating-point arithmetic can be implemented. Such would require a decent upgrade to the datapath.
3. Support exceptions. Add support for the EPC and the Cause registers.
4. Use the stack memory and support procedures.
5. Use dynamic branch prediction to decrease the branch time penalty.

Conclusion

This project went over two schemes for implementing a simple MIPS-32 processor that executes the Core instructions. The single-cycle version can execute an instruction every clock cycle. Such a scheme was impractical due to its timing inefficiency. The pipeline version uses different stages for which instructions can occupy simultaneously. The same components from the single-cycle scheme are used. Pipeline registers were introduced to define stages, and forwarding and hazard detection units were used to resolve dependencies throughput the pipeline. The conditional branch instruction is executed in the ID stage of the pipeline. For a program that contains dozens of instructions, the pipeline implementation can execute 1 instruction per cycle when the pipeline is full. The RISC-based MIPS ISA proves that easy and predictable implementation schemes can be built with ease. Because of the pipeline modularity, it is easier to debug problems.