Single Cycle MIPS Simulator

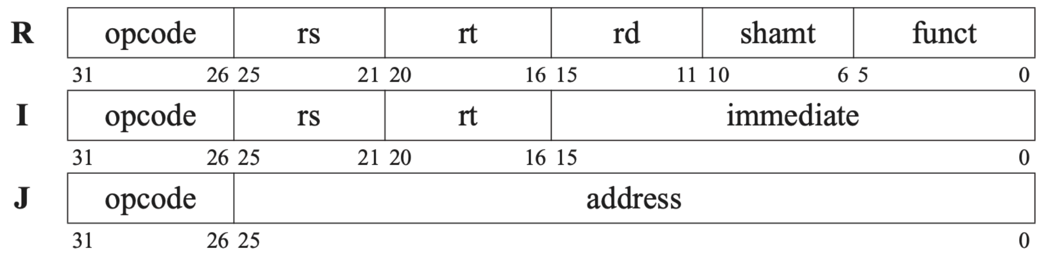
Nathaniel Vallen
32221415
Global SW Convergence
Free Days: 5

Introduction:

The purpose of the project is to create a single cycle MIPS simulator that is capable of running MIPS core instructions with some less commonly used instructions being excluded. A single cycle machine can be defined as a machine that runs all processes of executing an instruction every single clock cycle. In general, the overall process of executing an instruction in a single cycle machine consists of five steps, namely:

1. Instruction Fetch (IF)
2. Instruction Decode and Register Operand Fetch (ID/RF)
3. Execute/Evaluate Memory Address (EX/AG)
4. Memory Operand Fetch (MEM)
5. Store/Writeback Result (WB)

In instruction fetch stage (IF), the instruction pointed by the program counter (PC) in the instruction memory is fetched. Then, in instruction decode and register operand fetch stage (ID/RF), this instruction is decoded into several parts depending on the types of instructions. We can define the instruction type by reading the first six bits of the instruction, which is the opcode. Basic MIPS instruction format is as follows:



Generally, the opcode is passed onto the control box to determine which control signals are used and which are not. The source register (rs), target register (rt), and destination register (rd) indicate which registers to access to either read or write, depending on which input they are fed to. The registers take in four inputs, which are “Read register 1”, “Read register 2”, “Write register”, and “Write data”. Read registers 1 and 2 signify the register values to be read and passed onto the next stage. “Write register” marks which register should be updated with a new value. “Write data” means the updated value to be fed to the register to be updated (“Write register”). The shift amount (shamt) is also fed to the read register if the instruction processed is a shifting instruction (e.g. SLL/shift left logical). Function (funct) is passed onto the control box in case of opcode 0, or in other words, if the current instruction is an R-type instruction as R-type instructions are further identified by their function. Usually, the function bits will determine which operation should be performed by the ALU. An immediate value is not passed onto the registers but instead sign-extended from 16 to 32 bits. Lastly, an address is also processed further to become a jump address by shifting it by two bits and concatenating the first four bits to the beginning of the shifted address bits.

Next, in the execute/evaluate memory address (EX/AG) stage, we can use the previously fetched register operands to perform a specific operation in the ALU. Based on what instruction is being processed, the operation done in the ALU can differ too. Basically, the ALU is capable of performing arithmetic logical operations including OR, AND, NOR, NAND, XOR, ADD, SUB, etc. The calculation result of ALU can also be used as either an updated value of a target/destination register or an address to access in the data memory. In the memory operand fetch (MEM) stage, the value located in the previously calculated memory address is fetched. Since not all instructions require memory access, this stage is also, by nature, not always necessary in every instruction. Only memory accessing instructions such as load word (LW) or store word (SW) instructions need this stage. Finally, in the store/writeback result (WB) stage, the final value is stored in the target/destination register. The source of the value to be written is determined by the previous steps. In general, the two main sources of this value are data memory output and ALU result. However, in other few cases, there may be other special values to write. For example, the value to be updated to the register in a JAL (jump and link) instruction is PC + 8. It's noteworthy that not all instructions need each and every stage of the single cycle processing.

In some cases, bit encoding is required to proceed with the process. One of the cases is when we are passing an immediate value to an ALU. Since ALU takes in 32 bits input, the said immediate value should be sign-extended from 16 to 32 bits. Another case occurs when processing a jump or branch address. An address (instruction [25-0]) should be multiplied by four and concatenated with PC + 4 as explained previously. This is because the size of each instruction is 32 bits which equates to 4 bytes, meaning that every index in the instruction memory will be incremented by four per instruction. The same process is valid for an rs (source register) value taken to be an address in a branch instruction (excluding the concatenation).

The single cycle MIPS simulator uses multiple hardware components, mainly a program counter (PC), an instruction register, a data memory, 32 general purpose registers (GPR), 4 ALUs, 9 MUXs, and a control box for control signals. The complete datapath of this project is presented as such:

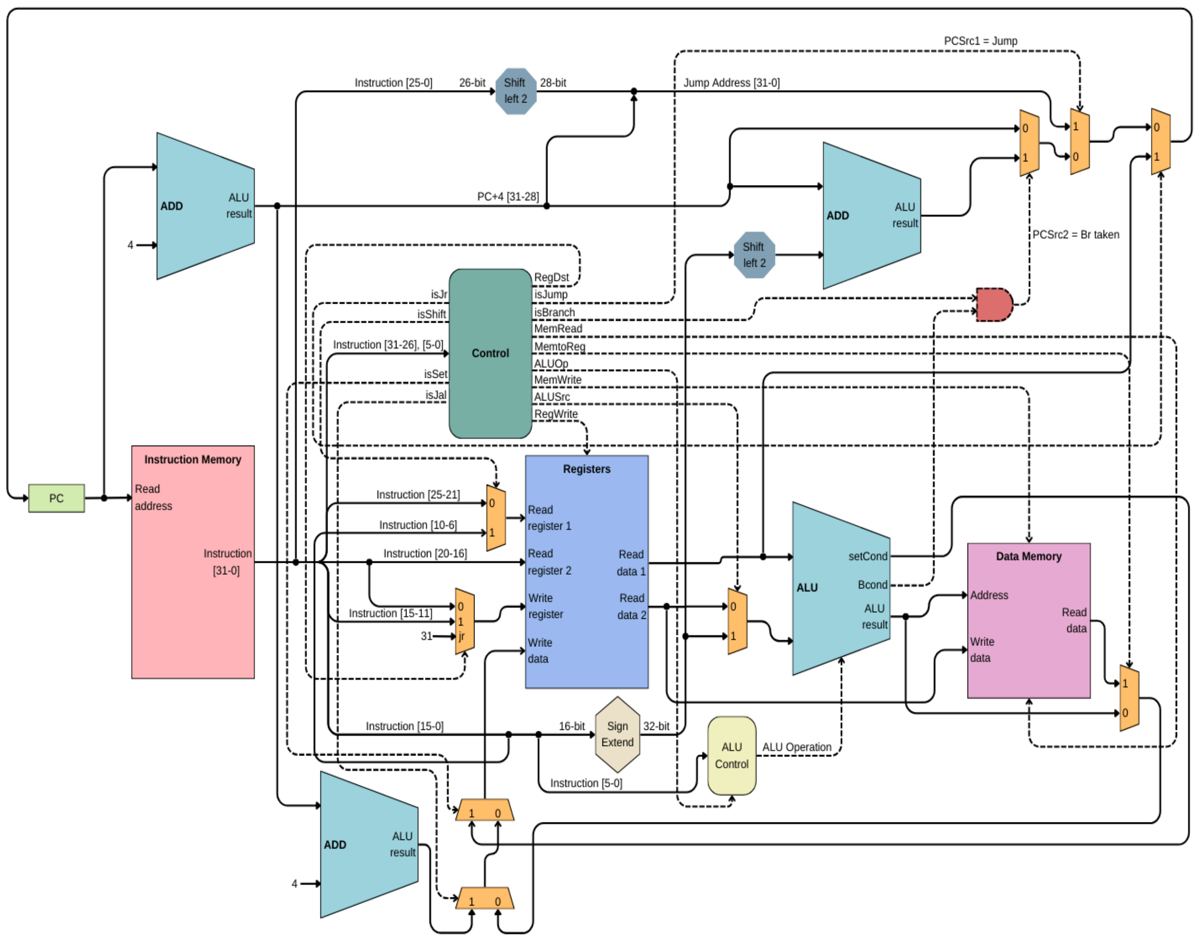


Figure 1. Full datapath of the single cycle MIPS simulator

Important Concepts & Considerations:

I. The following MIPS core instructions are excluded from this program:
   A. Load Byte Unsigned (LBU) [24]
   B. Load Halfword Unsigned (LHU) [25]
   C. Load Linked (LL) [30]
   D. Store Byte (SB) [28]
   E. Store Halfword (SH) [29]
   F. Store Conditional (SC) [38]

II. Clock cycle time is not set to a certain fixed value. In practice, the clock cycle time should adjust to the slowest instruction (LW/Load Word).

III. Control signal status for every instruction can be described as such:

Inst	Reg Dst	Mem Read	Mem to Reg	Mem Write	ALU Op	ALU Src	Reg Write	is Jump	is Branch	is Jr	is Shift	is Set	is Jal
J	0	0	0	0	-	0	0	1	0	0	0	0	0
JAL	2	0	0	0	-	0	1	1	0	0	0	0	1
BEQ	0	0	0	0	-	0	0	0	1	0	0	0	0
BNE	0	0	0	0	-	0	0	0	1	0	0	0	0
ADDI	0	0	0	0	‘+’	1	1	0	0	0	0	0	0
ADDIU	0	0	0	0	‘+’	1	1	0	0	0	0	0	0
SLTI	0	0	0	0	-	1	1	0	0	0	0	1	0
SLTIU	0	0	0	0	-	1	1	0	0	0	0	1	0
ANDI	0	0	0	0	‘&’	2	1	0	0	0	0	0	0
ORI	0	0	0	0	‘|’	2	1	0	0	0	0	0	0
LUI	0	0	2	0	-	0	1	0	0	0	0	0	0
LW	0	0	1	0	‘+’	1	0	0	0	0	0	0	0
SW	0	0	0	1	‘+’	1	0	0	0	0	0	0	0
SLL	1	0	0	0	‘<’	0	1	0	0	0	0	0	0
SRL	1	0	0	0	‘>’	0	1	0	0	0	0	0	0
JR	0	0	0	0	-	0	0	0	0	1	0	0	0
JALR	2	0	0	0	-	0	1	0	0	1	0	0	1
ADD	1	0	0	0	‘+’	0	1	0	0	0	0	0	0
ADDU	1	0	0	0	’+’	0	1	0	0	0	0	0	0
SUB	1	0	0	0	‘-’	0	1	0	0	0	0	0	0
SUBU	1	0	0	0	‘-’	0	1	0	0	0	0	0	0
AND	1	0	0	0	‘&’	0	1	0	0	0	0	0	0
OR	1	0	0	0	‘|’	0	1	0	0	0	0	0	0
NOR	1	0	0	0	‘-’	0	1	0	0	0	0	0	0
SLT	1	0	0	0	-	0	1	0	0	0	0	1	0
SLTU	1	0	0	0	-	0	1	0	0	0	0	1	0

IV. Apart from “ALURes” (ALU result), ALU can also produce two additional results, namely “setCond” (set condition) and “brCond” (branch condition). Based on the fulfillment of the condition, the value is set to either ‘1’ or ‘0’ (‘1’ if fulfilled, ‘0’ otherwise). Set condition is fulfilled if the value in the source register (rs) is smaller in comparison to either an immediate value (for SLTI or SLTIU) or a value in the target register (rt) (for SLT or SLTU). Meanwhile a branch condition is fulfilled if the values in the source register (rs) and target register (rt) are equal (for BEQ) or unequal (for BNE). Arithmetically, these comparisons can be done by performing a subtraction operation (equal if the result is zero, and vice versa).

V. Even though they are not used in the current test programs, the ALU is capable of performing two secondary operations, which are multiplication and division.

Unique Features:

- The code has a debug function which can be enabled by defining DEBUG_PC in the main.c file or by running -DDEBUG_PC with gcc compiler.
  - The debug function prints out the instruction count, opcode, and PC alongside any other related pieces of information according to the instruction type in a log file named “log.txt”.
- A function showStats() is available to display all register values for debugging or other purposes in case the user wishes to know the value of a specific register (this is currently disabled as a command).
- Optionally, the code can also display the read binary, reordered binary, opcode, and function of the current instruction (this is currently disabled as a command).
- The code is supposedly able to run JALR instructions. Unfortunately, due to the time limitations, I am unable to finish the modified fib binary file with JALR instructions, hence not being able to test it in time. However, the code for JALR is fully implemented and the datapath for JALR is sketched as follows:

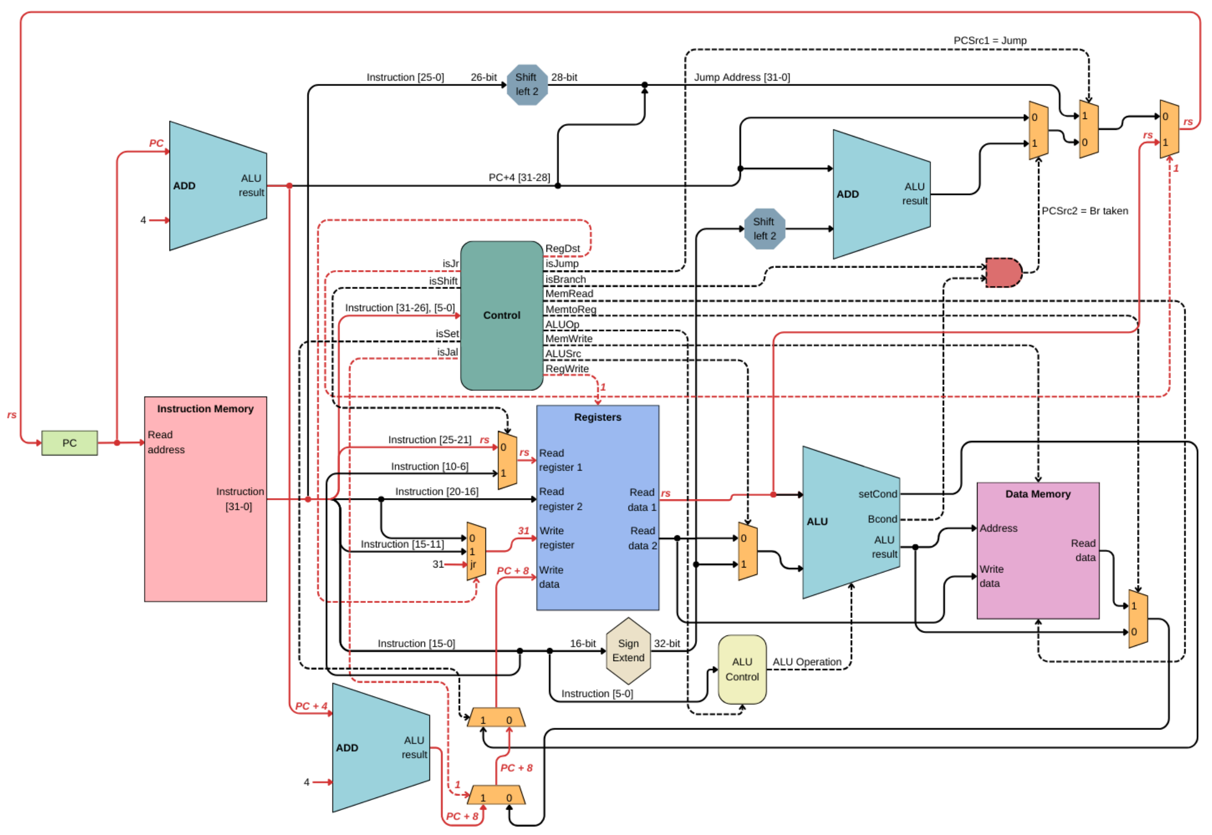
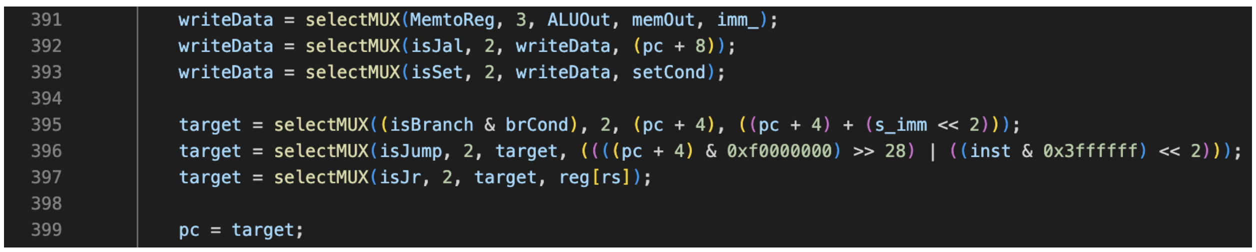


Figure 2. Datapath of JALR

Build Configuration & Environment

The code in this project was written in C programming language using Visual Studio Code IDE.

Screen Captures

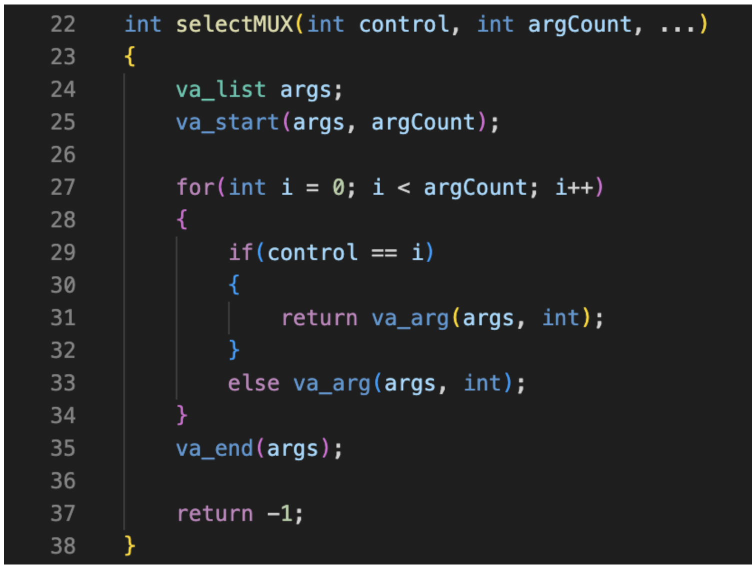


```
writeData = selectMUX(MemtoReg, 3, ALUOut, memOut, imm_);
writeData = selectMUX(isJal, 2, writeData, (pc + 8));
writeData = selectMUX(isSet, 2, writeData, setCond);

target = selectMUX((isBranch & brCond), 2, (pc + 4), ((pc + 4) + (s_imm << 2)));
target = selectMUX(isJump, 2, target, ((((pc + 4) & 0xf0000000) >> 28) | ((inst & 0x3ffffff) << 2)));
target = selectMUX(isJr, 2, target, reg[rs]);

pc = target;
```

Figure 3. Code snippet of datapath selecting section

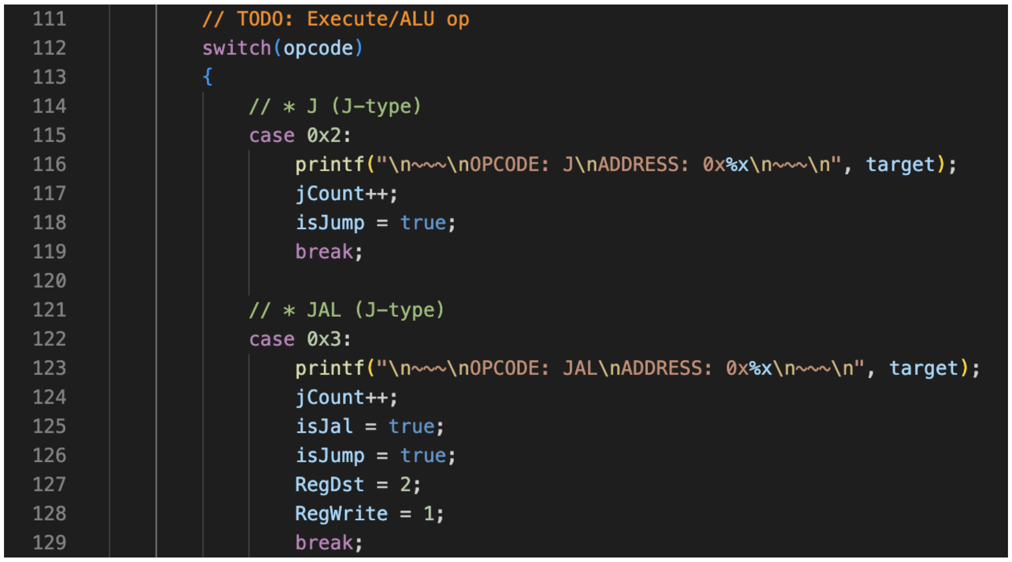


```c
int selectMUX(int control, int argCount, ...)
{
    va_list args;
    va_start(args, argCount);

    for(int i = 0; i < argCount; i++)
    {
        if(control == i)
        {
            return va_arg(args, int);
        }
        else va_arg(args, int);
    }
    va_end(args);

    return -1;
}
```

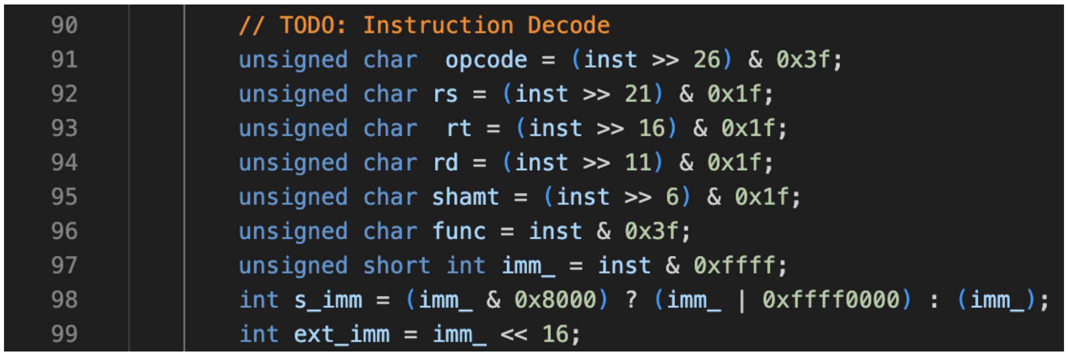
Figure 4. Code snippet of MUX input selecting function using variadic function



```c
        // TODO: Execute/ALU op
        switch(opcode)
        {
            // * J (J-type)
            case 0x2:
                printf("\n~~~\nOPCODE: J\nADDRESS: 0x%x\n~~~\n", target);
                jCount++;
                isJump = true;
                break;

            // * JAL (J-type)
            case 0x3:
                printf("\n~~~\nOPCODE: JAL\nADDRESS: 0x%x\n~~~\n", target);
                jCount++;
                isJal = true;
                isJump = true;
                RegDst = 2;
                RegWrite = 1;
                break;
```

Figure 5. Code snippet of execution stage



```c
        // TODO: Instruction Decode
        unsigned char  opcode = (inst >> 26) & 0x3f;
        unsigned char rs = (inst >> 21) & 0x1f;
        unsigned char  rt = (inst >> 16) & 0x1f;
        unsigned char rd = (inst >> 11) & 0x1f;
        unsigned char shamt = (inst >> 6) & 0x1f;
        unsigned char func = inst & 0x3f;
        unsigned short int imm_ = inst & 0xffff;
        int s_imm = (imm_ & 0x8000) ? (imm_ | 0xffff0000) : (imm_);
        int ext_imm = imm_ << 16;
```

Figure 6. Code snippet of instruction decode stage