COE-301

Implementing a Reduced Instruction Set Computer CPU

By

Muath F Alsubhi & Abdullah Hejazi

For

Dr. Saleh AlSlaeh

COE-301-55: Computer Organization

May 13, 2023

Abstract

This project will emphasize the design and implementation of a single cycle CPU which includes a program counter and its controller, arithmetic and logic unit, memory unit, control unit, and a file register.

- ***Objectives:***
  - ***Introduce Logisim and its simulator as a platform to build and test virtual circuits designs.***
  - ***Cooperations between team members to accomplish tasks and a project accordingly.***
  - ***Fully implement and develop a Pipelined 16-bit RISC processor with 16 bit instructions.***
  - ***Understand the different components of the Pipelined CPU***
- ***Introduction:***

In this project, We must build a 16-bit Instruction set reduced instruction set computer which will operate on Logisim as a platform, Such that we would have 2 bytes per instruction. Moreover, we would have 7 general purpose registers and a R0 of value 0, because of the frequent use of the value 0. Additionally, instructions includes three types of instruction R-type, I-type and J-type all of those instructions contain 5 bits located at the end for the opcode. First, R-type instructions contain 2 bits for the function call, 3 bits for the RT register (A), 3 bits for the RS register (B) and 3 bits for the RD destination register (C) and the 5 bits for Opcode. Using R-type instructions we can use all the operations in our ALU such as A+B=C or A XOR B = C etc. In addition, the I type instructions will have 5 immediate (either signed or unsigned) number inserted with the instruction and a register RT (A) and RS(B) such that we can use immediately instruction by providing one register and a number to preform an operation, such as the B = A + 5 or B = A ANDI 1 and other ALU operations also including the store and load instructions which uses the memory unit. Finally, the J-instructions which enables us to use instructions such as the LUI, JAL and Jump instructions which is important for loops. Moreover the J-type features an 11 bit sign extended immediate and 5 bits for the Opcode. The LUI stores explicitly in R1 and JAL stores explicitly in R7. Furthermore, we are required in this project to implement the above instructions and creating ALU, memory unit, controller unit, Program counter, program counter controller, and a second ALUb for immediate instructions. After successfully implementing a single cycle CPU, our computer can be further enhanced in terms of speed and cycles per instruction as we implement a pipelined CPU. The pipelined CPU will have intermediate registers that temporarily hold the value and allow components to be free after processing the instruction and allowing the next instruction to occupy it. The pipelined CPU will divide our CPU into 5 parts, first the instruction fetch, then the decoding stage, then the execution stage, then the memory stage and finally the writing back stage. Each stage will preform its designated part and wait until the previous stage passes values.

- ***Design and Implementation:***

Single Cycle CPU

Register File:

Our file register design is fairly simple where we have the register destinations, register A and register B as input. For the writing portion, we decode the register destination using a decoder such as from 0 to 7 (3 bits in binary) is decoded to 8 register on the right from R0 to R7. Then using BusW and writeEnable ,as input also, to write into the chosen destination. Additionally, the output of every register is connected to a multiplexer of A and similar design to B, where we use the Rs and Rt to choose an out from R0 to R7 depending on the input given to the register file unit. This design was chosen because of its simplicity and clarity. Moreover, for the JAL and JALR instructions we have added special support for them inside the file register, such that when we encounter their Opcode, we change the selection line of a multiplexer to write PC+1 that is calculated inside the main unit with some messy wiring to avoid many labels that are tedious.

Arithmetic Logic Unit:

Our ALU have a straight forward implementation where we have a multiplexer to choose the output of the given operation (4 bits, lower two is the function and upper 2 is from the opcode) is a selection line where all operation will still work in parallel. This design approach is because of simplicity and easy access to visualize and analyze each operation for easier debugging and enhancement in the future. In this unit we have bus A and B, and the function call as input, the unit includes 4 shifters, 3 comparators, two adders, XOR, AND, NOR and OR also a splitter to choose the lower 5 bits of the given bus B. Finally, for the output we have the result as output and as a designer choice, we have introduced an additional output of the COMP which gives the result of the comparison between A and B because the ALU works as combinational logic (not depending on clock) for later use the I-type branching instructions (beq, bnq, bge, blt) and special support for LUI.

Control Unit:

Our control unit approach we have a high memory dependency but highly effective and efficient. To elaborate, we have used two ROMs, the first is used to decode the given opcode into several outputs which are used into this form: BusWR(2), MemRed(1), MemWrite(1), ALuB(2), RegWirte(1) RegDis(2) a total of 9 bits, Also please note that the ALUB is used in the multiplexer to choose a specific BusB, imm5s, imm5z, or imm11s that will be explained later on. The second ROM, functions as an ALUOP decoder, where given some Opcode it will result in a hard-coded result of an alu operation to be used in ALU multiplexer explained earlier.

PC Control Unit:

This unit is important for I-type and J-type instructions where a special two bit functions will decide if the program counter is modified or not. Initially all of the PC+imm5 and PC+11imm and PC+1 is calculated in the main circuit. However, the PC controller output will decide via a multiplexer selection what one will be used into our multiplexer. The implementation choice in this circuit is very efficient as we have the unit taking input the opcodes of the branch, and taking the comparison result from the CPU between A, and B. Followingly, we have used a priority encoded such that if the branching flag which is an AND gate between each branch input opcode and its corresponding comparison condition are true then the result of the output is changed to 01 which means that our PC controller will change the next PC into PC+imm5, or if the jump instruction (J, JAL, JALR) opcode is received then it will immediately outputs a 10 to the PC source causing a PC+imm11 to the next program counter.

Main Unit:

This unit is just a combination of the previous discussed units. Where we will introduce the PC and we have a ROM memory that we write our program in hexadecimal then it will be fetched by the control unit, executed by the ALU and PC controller unit, then finally using memory write/read when necessary. A multiplexer with the register destination as a selector (for LUI and JAL commands). Also, another multiplexer to operate the immediate instructions via the ALUB as selection. And finally a multiplexer for writing memory in BusW or not for load instruction. This approach is fairly simple and easier to maintain. Also we have used tunneling to be visually appealing throughout the design and previous units.

Pipelined CPU

Intermediate Register File:

The core part of a pipelined CPU that enable the processing of multiple instructions at the same time. Each stage out of the 5 stages will have a register at the end of the stage to store the result value. As an effect, in the next clock cycle the resource for that stage is now processing the next instruction while that stored value is now passed into the next value. Subsequently, every stage will do a similar procedure until reaching the last stage. Such approach will create data, structure, and control hazards which are dealt with through other units presented in this design with intelligent solutions such as forwarding, stalling, and killing (ignoring) instructions that are later explained. To elaborate, in the first stage we have the instruction register that stores the current instruction and awaiting the next one. Following, the result of that register is then passed to the decode stage where we will get Rs, Rt, Rd, immediate values, ALU operation and others that are explained in the single cycle part. In addition, the next program counter is also stored for for branching calculation results. Additionally, that calculation is also stored in a register, since our result of branching cannot be decided until the execution stage. Moreover, the main control unit output is stored in a register that waits the execution stage to provide its outputs correctly without delay for correct results except for the register destination code that is stored separately and delayed for 3 clock with 3 registers cycles to reach the writing back stage. Additionally, the results of the register file which is the busA, busB, and the immediate 5 bit values are stored in a register for the execution stage. Now, in the execution stage we will pass the results and the data writing bus into separate registers for the memory stage. Finally, we will have a register at the end of the memory stage to store the final data output which may or may not be used in writing back the data depending on the instruction. Also the main control execution register have a register for delaying the signals and enables of the memory stage, also the writing back destination enabler is delayed until the last, writing back, stage.

Data Hazards:

Initially, some instructions may have dependency. For example, our first instruction may calculate the results of A + B = C that is calculated and written after 5 cycles. and the following instruction have a need for the value C on the second stage which will result in getting an incorrect C value. Such dependencies solution is to forward the ALU result to the busA, or busB of the next instruction by matching the Ra, or Rb with Rd of the previous instruction. Hence, we need to store the Rd of the previous instruction in a register, and then compare that value with the Ra and Rb of the next instruction, if they match we need to forward or stall in a special case then forward if the depends is after the memory stage then forward that value to the next instruction in simple terms matching the following if statements:

```
If ((Rs != 0) and (Rs == Rd2) and (EX.RegWr)) ForwardA = 1
Else if ((Rs != 0) and (Rs == Rd3) and (MEM.RegWr)) ForwardA = 2
Else if ((Rs != 0) and (Rs == Rd4) and (WB.RegWr)) ForwardA = 3
Else ForwardA = 0
If ((Rt != 0) and (Rt == Rd2) and (EX.RegWr)) ForwardB = 1
Else if ((Rt != 0) and (Rt == Rd3) and (MEM.RegWr)) ForwardB = 2
Else if ((Rt != 0) and (Rt == Rd4) and (WB.RegWr)) ForwardB = 3
Else ForwardB = 0
```

As our designer choice, in the CPU the previous rd is stored inside a register, then our unit will take the rd, ra and rb as input, compare with an AND gate and subtractor then if the result is 1 (true) then we will proceed with changing the busA or busB as per the if statements above. Such that before the busA and busB register we will have a 2 bit multiplexer with the forwarding conditions as selector lines and choosing from either forwarding what value and register Also we have used a priority encoder for that task. Additionally, for the stalling part, we will use a flag for the memory read from the decoding stage, if that is true and there is a data dependency that will be hazardous. Subsequently, we will compare the previous rd with the MemRd flag and our forwarding condition. If all is true hence we need to stall a cycle such that our results are correctly forwarded.

Control Hazards:

In a branching instruction we have two possible outcomes, the first is that we do not take the branch which then, we can just continue with the instruction as normal and no hazard will appear. However, if we decide to take the branch, that will result in great consequences in the CPU, since we can only get the result of the comparison in the execution stage which changes our original implementation in the PC controller into a more sophisticated one. To elaborate, there will be an instruction that is inside the decoding stage and another instruction waiting in the fetching stage. Subsequently to taking a branch, the address will change and the following to the branch instruction shouldn't be passed to our CPU. Hence we need to clear the instruction from the registers by a multiplexer (kill) that the selector is the killing output from the PC control unit that select between actual instruction and 0, if the branch is taken, (AND gate between the branch opcode and the zero flag of comparison from the CPU) then we will pass a signal to the register in the fetch stage to be 0 and the register after the main control unit to feed 0 and clearing the presence of the two instruction in the CPU. Additionally, if we encounter a jump, jump, link and jump and link register instructions, we need to immediately kill the next instruction by only feeding a kill to the fetch stage. Hence ensuring that our branching is correct and eliminated the hazard. As a special designer choice, we have used a passed the branch opcode from the main control unit to the PC controller then used a and gates between every branch opcode and its zero comparison flag. Our designer choice is fairly simple, if the branch condition and opcode is true through an AND gate, our killer1 and killer2 correspond to 1 and killing the next 2 instruction. If it is a jump instruction it will pass 1 into the first killer that kills the next instruction and the PC controller will proceed with jumping. However, with the jumping instruction we will only change the PC controller if the branching result is true.

• *Simulation and Testing:*

Single Cycle CPU

1- Counting the number of 1

ori R2, R0, 15	**7a09**	**Setting R2 to 15 such that our loop will run 16 times**
ori R4, R0, 11	5c09	This will be our variable X, it will count the number of 1s in the R4 register
LOOP		
ANDI R5, R4, 1	0d8a	This AND will result in given us the lowest bit in the register
add R3, R3, R5	1d60	The register R3 (initially 0) will hold the number of 1s in the register.
srli R4, R4, 1	0c8d	After doing it we will shift X (R4) one bit to the right logically
addi R2, R2, -1	fa44	Reducing our count variable
bne R2, R0, LOOP	e053	If the count variable isn't eq to 0, branch out (please notice this is a do while loop)
sw R3, 0(R0)	0311	Finally storing the result in memory for displaying
End		
J end	0018	To keep the program at the last instruction

2- R-type

lui 1	**0037**	**Setting R1 = 32**
ori R3, R1, 10	5329	Setting R3 = 42
add R4, R3, R1	2160	Add R3 + R1 and store in R4, R4= 74
sub R5, R4, R3	6b80	Subtract R4-R3 and stor in R5,

		R5=32
sw R4, 0(R0)	0411	Store 74 in the memory at position 0
sw R5, 1(R0)	0d11	Store 32 in the memory at position 1
SLTU R5, R4, R5	ed80	Do a comparison as unsigned and set R5 to 1 if R4 > R5, R5 = 0
XOR R5, R5, R4	2ca1	XOR R5 with R4, R5 = 74
SLT R5, R5, R5	ada0	Do a comparison and set R5 to 1 if R5 < R5, else set to zero, R5 = 0
OR R6, R5, R4	74a1	OR R5 with r4 and store in R6, R6 = 74
AND R6, R6, R1	b1c1	AND R6 with R1, R6 = 0
NOR R6, R6, R0	f0c1	NOR R6 with R1, R6 = -1
SRLi R1, R1, 2	112d	Shift R1 two bits to the right, logic extension, R1 = 8
addi R1, R1, 6	3124	Add R1 + 6 and store in R1, R1=14
SRL R6, R6, R1	71c2	Shift R6 14 bits to the right, logic extension, R6 = 3
sw R6, 2(R0)	1611	Store 3 in the memory at position 2
SLL R6, R6, R6	36c2	Shift R6 three bits to the left, R6 = 24
addi R1, R1, -11	a924	Subtract R1 – 11, R1 = 3
SRA R6, R6, R1	b1c2	Shift R6 3 bits to the right, arithmetic extension, R6 = 3
addi R1, R1, 11	5924	Add R1 + 11, R1 = 14
ROR R6, R6, R1	f1c2	Rotate right 14 bits , R6 = 12
J exit	0018	Indefinite jump

3- I-type

lui 1	**0037**	**Setting R1 = 32**
addi R3,R1,10	5324	Add R1 + 10, R3 = 42
SLTI R4,R3,-50	7466	R4 will have the value 0 because R3 is greaters than -50
SLTIU R5,R3,50	9567	R5 will have the value 1 because R3 is less than 50
XORI R6, R0, 3	1e08	R6 will have the value 3 because 0 xor y = y
ORI R7 , R5 , 1	0fa9	R7 will have the value 1 because 0 or 1 = 1
ANDI R2 , R7 , 0	02ea	R2 will have the value 0 because 1 and 0 = 0
NORI R1 , R3, -10	b16b	R1 will have the value -63
SLLI R5,R5,1	0dac	R5 SHOULD HAVE THE VALUE 2
SRLI R3,R3,2	136d	R3 will have the value 21
SRAI R1,R1,2	112e	R1 = -16

RORI R7,R7,14	77ef	r7 will have the value 4
J exit	0018	Indefinite loop

4- branches and jump

jal end2	**0904**	**This will jump to end2 and store pc+1 in R7**
end2:	1a24	R2 = 4
beq R2, R0, end3	0310	This is not take
bne R0, R0, end3	2260	This is not take
bge R0, R6, end3	6b81	This branch will be taken
blt R6, R0, end3	30b3	This branch is never reached
lui 1	34a0	# R1 = 32
j finalEnd	0058	Jump to final end
end3:	2da0	
addi R1, R0, 19		This addition will not be excuted
finalEnd:	1012	
j finalEnd	2c04	Indefinite loop

5- bubble sort, please wait it until over

addi R1, R0, 15	**0059**	**n**
addi R7, R0, -1	0018	R7= -1 constant
	2104	
loopi:	1000	# we need to reset our R3
add R3, R0, R0	1800	# j = 0
loopj:	0c04	# assume R4 and R5 to be our arr[j] and arr[j+1]
lw R4, 0(R3)	fe84	# arr[j]
lw R5, 1(R3)	02d0	# arr[j+1]
slt R6, R4, R5	0590	#R6 = 1, R4 < R5 . hence if 5 is less, R6 is zero.
beq R6, R0, swap	2554	# R4 < R3
j else	1d52	
swap:	05d1	
sw R5, 0(R3)	0291	
sw R4, 1(R3)	0c84	
else:		
addi R3, R3, 1	5320	# j = j+1
bne R3, R1, loopj	ba94	# when j = n, we don't loop # loop j is done now
addi R2, R2, 1	0b64	#increment i
bne R1, R2, loopi	a174	#i == n, stop loo
end:		
J end	07f6	

Note: Our array is the first 16 bit

Pipelined

1- data hazard

addi R1, R0, 1	**0904**	**R1 = 1**
addi R2, R1, 3	1a24	R2 = 4
lw R3, 0(R0)	0310	R3 = 5
add R4, R3, R2	2260	R4 = 9
or R5, R4, R3	6b81	R5 = 13
BNE R5, R0, exit	30b3	This branch should be taken R5 != R0
add R6, R5, R4	34a0	This addition will not work be cause of the brnack
j next	0058	Jump to next
add R5, R5, R5	2da0	R5 should not be excuted and R5 = 13 as before
next:		R3 will have the value 21
BEQ R0, R0, next2	1012	This branch should be taken since R0 = R0
addi R4, R0, 5	2c04	This addition will not be work because of the jump
next2:		Indefinite loop
Exit:		
J Exit	0018	Indefinite loop

- ***Team Work:***

Single Cycle CPU

The following table will summarize the team work on every task:

File Register	Muath 90% Abdullah 10%
ALU	Muath 65% Abdullah 35%
Control Unit:	Abdullah 90% Muath 10%
PC Control Unit	Muath 70% Abdullah 30%
Main Unit:	Abdullah 50%, Muath 50%
Report:	Muath 75% Abdullah 25%

Pipelined CPU

Intermediate Registers	Muath 85% Abdullah 15%
Data Hazards	Muath 0% Abdullah 100%
Control Hazards	Muath 80% Abduallah 20%

Report:	Muath 80% Abdullah 20%

- ***Conclusion:***

In conclusion, this project is a cumulation of knowledge that a student must deeply understand how an instruction set functions, how digital logic operates, and how programming is implemented. Furthermore, in this project we learned that we must individually design every component of a CPU independently and verify its functionality. Moreover, then we need to perform the 4 steps on a single cycle CPU which is fetching, when we use the control unit to decode our opcode into inputs for other components and fetch the right registers for execution. then , we execute using the ALU and/or memory where we either write our result from the execution inside our register file or into the memory RAM. Followingly, to further improve our CPU performance in larger application we have implemented a pipelined 5 stage CPU that made our CPU superior and faster than the single cycle by far. However, we had to solve hazards as we have implemented solutions for data dependency, branching and jumping hazards In addition, a deep understanding of the CPU architecture is important to be able to replicate it and built it ourselves based on our requirements and cost available. Furthermore, Logisim proved to be a powerful tool and is used as a platform for digitally simulate our CPU. Finally, we found this project to be interesting and challenging.