Team Yellow

Memory to Memory

Wenze Ma
Sadie Park

Table of Contents

Overview

Our design is using a **memory-memory architecture**, which we are going to apply to the processor. This processor will have a small number of instructions because it does not have to load values to the registers from the memory, or save values of registers to the memory. This will keep the program at a relatively shorter length. However, each instruction will have more bits in comparison to the other architectures.

Registers

There are no registers available for programmers because everything is stored in the memory. But there are three registers reserved for specific purposes.

REGISTER	NAME	NUMBER	USE
$pc	Program Counter	00	It stores the address of the program that is currently executing.
$sp	Stack Pointer	01	It stores the address of the top address of the stack
$ra	Return Address	10	It stores the return address of a procedure

Instructions

We will have four types of instructions in our architecture. Each of our instructions will be 64-bit long. Because it needs to store three memory addresses, and each memory address is 16-bit long, we think a 64-bit instruction is plausible to store all three addresses plus the opcode and function code.

M-Type (Memory)

op (8 bits)	ms (16 bits)	mt (16 bits)	md (16 bits)	unused (8 bits)

M-Type instruction deals with arithmetic operations between two values in the memory. It takes an opcode, three memory addresses, and an 8-bit immediate. The operation is determined by opcode. It will apply the arithmetic operation to the two values from the source memory address (ms and mt) and store the result of the operation to the destination memory address (md).

Syntax:
op md, ms, mt

Operations:

1. **add:** Get the values from the two source memory addresses, and add them, and put the result to the destination memory address
2. **sub:** Get the values from the two source memory addresses, and subtract the value stored in mt from the value stored in ms, then put the result to the destination memory address
3. **and:** Get the values from the two source memory addresses, and apply bitwise operation “and” to the two values, then store the result to the destination address.
4. **or:** Get the values from the two source memory addresses, and apply bitwise operation “or” to the two values, then store the result to the destination address.
5. **j:** Jump to the md unconditionally.
6. **jal:** Set $ra to $pc + 8, and jump to md unconditionally. (RELATIVE)

I-Type (Immediate)

op (8 bits)	ms (16 bits)	mt (16 bits)	imm (16 bits)	unused (8 bits)

I-Type instruction deals with operations containing immediate values. It takes an opcode, two memory addresses, and an immediate value. It derives the operation from the opcode, and applies the operation to the value from the the source memory address (ms) and the immediate value (imm). After the operation, it will store the result of the operation to the destination memory address (md).

Syntax:
op md, ms, imm

Operations:

1. **addi:** Get the value stored in the memory address ms, and add the immediate value to it, then store it to the memory address mt.
2. **ori:** Get the value stored in the memory address ms, and apply the bitwise “or” operation to the value and the immediate value, then store the value to the memory address mt.
3. **andi:** Get the value stored in the memory address ms, and apply the bitwise “and” operation to the value and the immediate value, then store the value to the memory address mt.
4. **sll:** Get the value stored in the memory address ms, and shift left with regards to the immediate value, and store it to mt.
5. **srl:** Get the value stored in the memory address ms, and shift right with regards to the immediate value, and store it to mt.
6. **beq:** If the value stored in memory address ms is the same as the one stored in mt, branch to the current address plus the immediate plus 4.
7. **bne:** If the value stored in memory address ms is not the same as the one stored in mt, branch to the current address plus the immediate plus 4.
8. **bgt:** Branch to $pc + 4 + imm if the value of ms is greater than the value of mt
9. **blt:** Branch to $pc + 4 + imm if the value of ms is less than the value of mt
10. **bge:** Branch to $pc + 4 + imm if the value of ms is greater than or equal to the value of mt
11. **ble:** Branch to $pc + 4 + imm if the value of ms is less than or equal to the value of mt
12. **jr:** Jump to the address stored in $ra.
13. **save:** It can save the value in the memory address on the stack at the position of $sp + imm
14. **load:** It changes the value of the address to the value at $sp + imm.
15. **allocate:** It moves $sp down by the immediate value
16. **free:** It moves $sp up by the immediate value
17. **li:** It sets the value of the memory address to the immediate
18. **rtm**: It sets the value of memory address mt to the value of the memory address ms, and set $pc to $ra
19. **rti**: It sets the value of memory address mt to the immediate, and set $pc to $ra
20. **sr**: It saves $ra on the stack at $sp + immediate
21. **lr**: It loads $sp + immediate to $ra

B-Type (Branch)

op (8 bits)	ms (16 bits)	imm (16 bits)	imm2 (16 bits)	unused (8 bits)

B-type instruction handles the branch operations that involve comparison with an immediate. It takes an op, a memory address, and two immediates.
Syntax:
op ms, imm, imm2

Operation:

1. **beqi:** If the value stored in memory address ms is the same as imm2, branch to the current address plus the imm plus 4.
2. **bnei:** If the value stored in memory address ms is not the same as imm2, branch to the current address plus the imm plus 4.
3. **bgti:** Branch to $pc + 4 + imm if the value of ms is greater than imm2
4. **blti:** Branch to $pc + 4 + imm if the value of ms is less than imm2
5. **bgei:** Branch to $pc + 4 + imm if the value of ms is greater than or equal to imm2
6. **blei:** Branch to $pc + 4 + imm if the value of ms is less than or equal to imm2

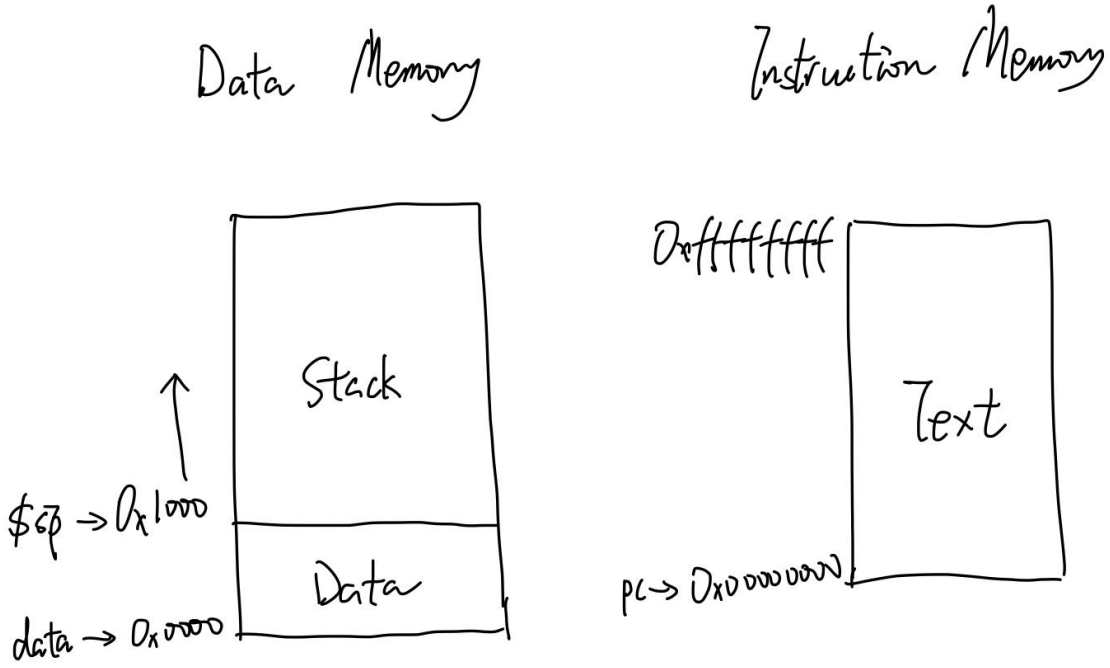
Symbolic Description

NAME, MNEMONIC		FOR-MAT	OPERATION	OP (0x)
Add	add	M	M[md] = M[ms] + M[mt]	00
Subtract	sub	M	M[md] = M[ms] - M[mt]	01
And	and	M	M[md] = M[ms] & M[mt]	02
Or	or	M	M[md] = M[ms] | M[mt]	03
Branch On Equal Immediate	beqi	B	if (M[ms] == Imm) $pc = $pc + 4 + Imm2	04
Branch On Not Equal	bnei	B	if (M[ms] != Imm) $pc = $pc + 4 + Imm2	05
Branch On Greater Or Equal Immediate	bgei	B	if (M[ms] >= Imm) $pc = $pc + 4 + Imm2	06
Branch On Greater Immediate	bgti	B	if (M[ms] > Imm) $pc = $pc + 4 + Imm2	07
Branch On Less Or Equal	blei	B	if (M[ms] <= Imm)	08

Immediate			$pc = $pc + 4 + Imm2	
Branch On Less Immediate	blti	B	if (M[ms] < Imm) $pc = $pc + 4 + Imm2	09
Branch On Equal	beq	I	if (M[ms] == M[mt]) $pc = $pc + 4 + Imm	0A
Branch On Not Equal	bne	I	if (M[ms] != M[mt]) $pc = $pc + 4 + Imm	0B
Branch On Greater	bgt	I	if (M[ms] > M[mt]) $pc = $pc + 4 + Imm	0C
Branch On Greater Or Equal	bge	I	if (M[ms] >= M[mt]) $pc = $pc + 4 + Imm	0D
Branch On Less Or Equal	ble	I	if (M[ms] <= M[mt]) $pc = $pc + 4 + Imm	0E
Branch On Less	blt	I	if (M[ms] < M[mt]) $pc = $pc + 4 + Imm	0F
Allocate Space On Stack	allo	I	$sp = $sp - Imm	10
Free Space On Stack	free	I	$sp = $sp + Imm	11
Load	ld	I	M[mt] = R[$sp + Imm]	12
Load Immediate	li	I	M[mt] = Imm	13
Load Return Address	lr	I	$ra = R[$sp + Imm]	14
Add Immediate	addi	I	M[mt] = M[ms] + Imm	15
And Immediate	andi	I	M[mt] = M[ms] & Imm	16
Or Immediate	ori	I	M[mt] = M[ms] | Imm	17
Shift Left Logical	sll	I	M[mt] = M[ms] << Imm	18
Shift Right Logical	srl	I	M[mt] = M[ms] >> Imm	19
Save	sv	I	R[$sp + Imm] = M[ms]	1A
Save Return Address	sr	I	R[$sp + Imm] = $ra	1B
Return Memory	rtm	I	M[mt] = M[ms] $pc = $ra	1C
Return Immediate	rti	I	M[mt] = Imm $pc = $ra	1D

Jump	j	I	$pc = Imm	1E
Jump And Link	jal	M	$ra = $pc + 4 $pc = md	1F
Jump Return	jr	I	$pc = $ra	18
Finish	fn	I		19

Memory Allocation



Assembly Translation of RelPrime

```
1300000004000200,
1000000000000200,
1B00000000000000,
1A00010000000200,
1700040002000000,
1F00000000000E00,
1200000001000200,
0400070001000200,
1500040004000100,
1E00000000000300,
1400000000000000,
1100000000000200,
1700040007000000,
1900000000000000,
0500010000000100,
1C00020007000000,
0400020000000500,
0E00010002000200,
0100010002000100,
1E00000000001000,
0100020001000200,
1E00000000001000,
1C00010007000000;
```

```
RELPRIME:   li T0, 2
            allo 4
            sr 0
LOOP2:      sv A0, 2
            ori A1, T0, 0
            jal GCD
            ld A0, 2
            beqi V0, 1, RETURN2
            addi T0, T0, 1
            j LOOP2
RETURN2:    lr 0
            free 2
            ori V0, T0, 0
            fn
GCD:        bnei A0, 0, LOOP
            rtm V0, A1
LOOP:       beqi A1, 0, RETURN
            ble A0, A1, CASE2
CASE1:      sub A0, A0, A1
            j LOOP
CASE2:      sub A1, A1, A0
            j LOOP
RETURN:     rtm V0, A0
```

Close Group

#Memory.v ●

Users > mawenze > Documents > Rose-Hulman > CSSE232 > #Memory.v

```
#Memory
0000    ZERO    #stores the value of zero
0001    A0      #stores the value of the first argument
0002    A1      #stores the value of the second argument
0003    A2      #stores the value of the third argument
0004    T0      #stores the tempory value
0005    T1      #stores the tempory value
0006    T2      #stores the tempory value
0007    V0      #stores the first return value
0008    V1      #stores the second return value
#stack
1000
1001
```

RTL

Step	M-type	beq/bne/bgt/blt/bge/ble	B-type
Inst Fetch	IR = IMem[PC] PC = PC + 2		
Inst Decode	A = DMem[IR[55-40]] B = DMem[IR[39-24]] ALUOut = PC + IR[23-8]		A = DMem[IR[55-40]] Imm = IR[39-24] ALUOut = PC + IR[23-8]
Execution	ALUOut = A op B	if (A == B) then PC = ALUOut	if (A == Imm) then PC = ALUOut
Mem	DMem[IR[23-8]] = ALUOut		

Step	j	jal	allo/free	jr
Inst Fetch	IR = IMem[PC] PC = PC + 2			
Execution	PC = IR[23-8]	$ra = PC PC = IR[23-8]	$sp = $sp +/- IR[23-8]	PC = $ra

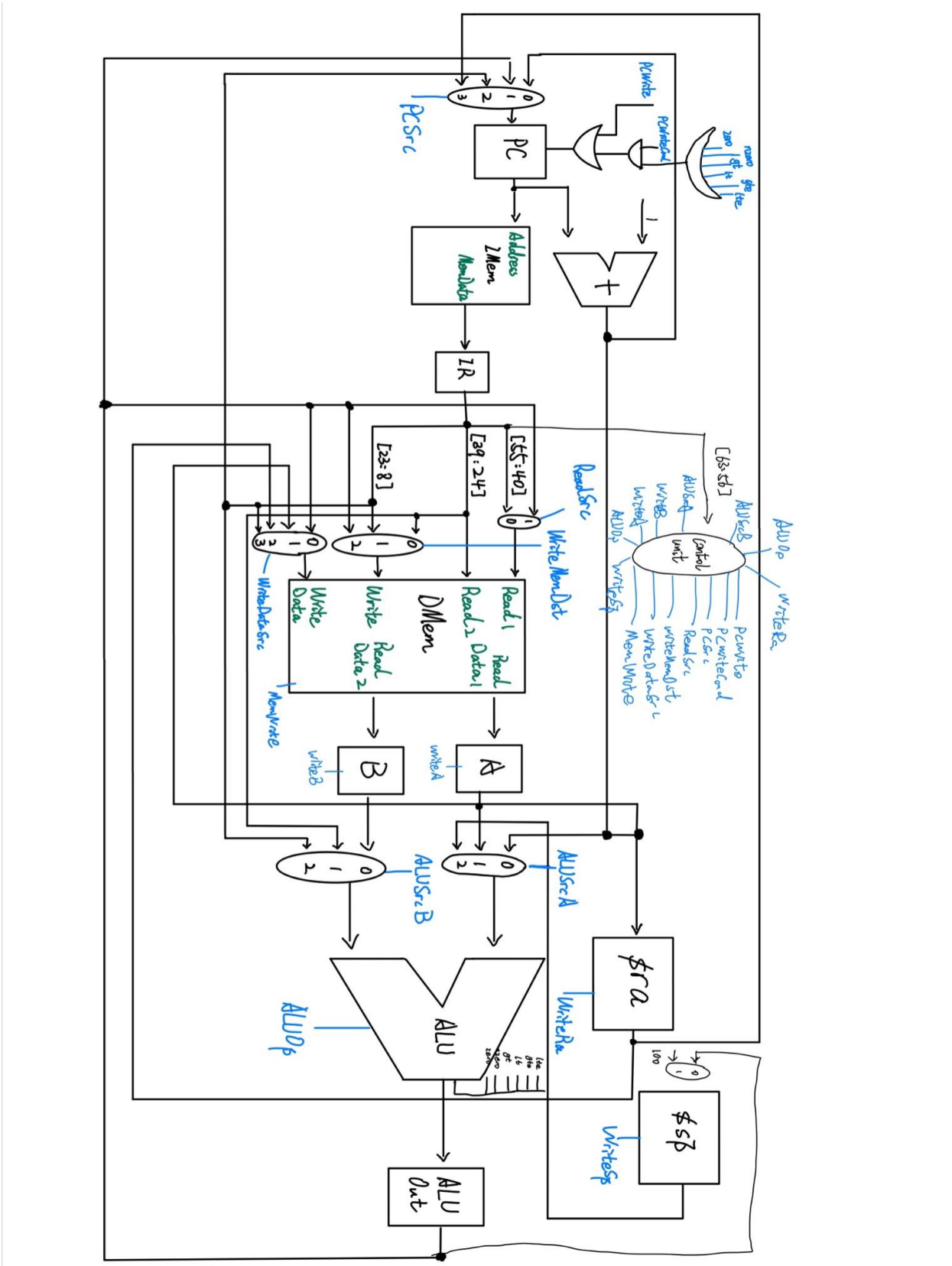
Step	I-type	sv	sr	rtm	rti
Inst Fetch	IR = IMem[PC] PC = PC + 2				
Inst Decode	A = DMem[IR[55-40]] Imm = IR[23-8] ALUOut = R[$sp] + Imm				
Execution	ALUOut = A op Imm	DMem[ALUOut] = A	DMem[B] = $ra	DMem[IR[39-24]] = A PC = $ra	DMem[IR[39-24]] = Imm PC = $ra
Mem	DMem[IR[39-24]] = ALUOut				

Step	ld	li	lr
Inst Fetch	IR = IMem[PC] PC = PC + 2		
Inst Decode	Imm = IR[23-8] B = R[$sp] + Imm A = DMem[B]		
Execution	DMem[IR[39-24]] = A	DMem[IR[39-24]] = Imm	$ra = A

Components

	Input Signal	Output Signal	Control Signal	Description
ALU	A (16 bits) B (16 bits)	ALUOut (16 bits) Overflow (1 bit)(Not necessary)	ALUOp (2 bits)	It takes two inputs A and B and performs an arithmetic operation that is given by ALUOp and produce 16bits outcome. If the ALUOut is bigger than 16 bits, it produces 1 bit overflow.
Instruction Memory	PC Address (16 bits)	IR (64 bits)		It reads the instructions from the PC and returns IR.
Data Memory	Read 1 (16 bits) Read 2 (16 bits) Write (16 bits) Write Data (16 bits)	Read Data 1 (16 bits) Read Data 2 (16 bits)	WriteMemDst (2 bits) WriteDataSrc (2 bits)	It takes two addresses to read from the memory and one address to write to the memory, and the actual data to write.
Controller	OP (8 bits)	PCSrc (2 bits) ReadSrc (1 bit) WriteMemDst (2 bits) WriteDataSrc (2 bits) ALUSrcA (2 bits) ALUSrcB (2 bits) WriteRa (1 bit) WriteSp (1 bit) ALUOp (2 bits)		Control multiplexers
PC	PC (16 bits)			It takes new address data when clock goes 1 and send out address of code that it need to perform.
PC Adder	PC (16 bits) Immediate value 2	PC		It takes the current PC address in and increment the PC address 8.

Data Path


PCSrc
PCWrite
PC
Address
IMem
MemData
IR
+
1
[23:8]
[29:24]
[15:40]
RegSrc
WriteMemDst
Write
Data
DMem
Write
Read
Data 2
Read 1
Read 2
Data 1
Read
MemWrite
WriteDataSrc
B
A
WriteB
WriteA
ALUSrcB
ALUSrcA
ALU
ALUOp
$ra
WriteRa
$sp
WriteSp
ALU
Out
Control
Unit
ALUOp
WriteRa

How to implement components:

	Implementation	Test
ALU	Apply the appropriate operation to the two inputs, and store the result to ALUOut	Test whether the ALUOut is correct based on the inputs and ALUOp
Instruction Memory	Just a memory file that can fetch data at the given address.	Not sure. Maybe check whether it gives correct value at a specific address.
Data Memory	Read 1 (16 bits) Read 2 (16 bits) Write (16 bits) Write Data (16 bits)	
Controller	Set the control signals to the appropriate value based on the opcode	Check whether it sets the control signals to the expected values.
PC		
PC Adder	A simple ALU that performs add operation	Check whether it correctly adds 2 to the PC

Control signal

	Input	Description
PCSrc 2bit	0 : PC + 1 1 : PC + ALUOut 2 : PC + IR[23 : 8] 3 : $ra	Normal incrementing For branch/branch imm For j and jal For jr, rtm (returning back)
WriteMemDst 2bit	0 : IR[39-24] 1 : IR[23-8] 2 : ALU OUT	mt md
MemWrite	0: Not wiritng 1: Writing	
WriteDataSrc 2bit	0 : ALU out 1 : A 2 : $ra 3 : IR[23-8]	The value in the memory of mt Immediate value

ALU OP 4bit	0 : A + B 1 : A - B 2 : A & B 3 : A | B 4 : branch	Add Sub And Or It changes the wire of zero, nzero, gt, gte, lt, lte based on the opcode. (e.g. if op tries to branch equal, and the two input values are equal, it will change zero to 1)
ALU srcA 2bit	It selects what goes into A 0 : PC + 1 1 : A 2 : $sp	Output of Dmem
ALU srcB 2bit	It selects what goes into B 0 : B 1 : IR[39:24] 2 : IR[23:8]	Output of Dmem mt Immediate value
ReadSrc 1bit	0: IR[55:40] 1: ALUOut	ms
WriteA/B 1bit	0: Cannot write to register A/B 1: Write to register A/B	
WriteRa/WriteSp 1bit	0: Cannot write to the register 1: Write to register	
PCWrite	0: Cannot write to PC 1: Write to PC	
PCWriteCond	0: Not trying to branch 1: Attempt to branch	

Integration Plan

1. First, we'll implement the data memory and the instruction memory with input and output. These will be the essential part of the datapath since we are using Memory-Memory architecture. We will test these components by storing the data in the .coe file, and see whether we can get the same data in the output. The instruction memory will be very similar to the one we used in lab 7. But the data memory will have two inputs.
2. Then, we'll implement the 16-bit PCAdder. One input will be the PC address and the other input will just be 1 because Xilinx is word based. For testing, we'll just check whether it performs the adding actions correctly.
3. Then, we'll implement the 16-bit ALU. This ALU will be complex, and it should perform the arithmetic operations based on the ALUOp. For testing, we'll check whether it gives the correct output based on the opcode.

4. Then, we'll implement a 16-bit PC register. It will have an input and an output. It also accepts a control signal to see whether it needs to change to the value based on the branch conditions. The testing will just check whether it will change based on different signals.
5. Registers for $ra and $sp. They are the same as the PC register. They also have one input, one output, and a control signal to indicate whether their value should be changed. The testing plan is the same as the PC register.
6. Registers for A, B, and ALUOut. They are similar to the other registers, except they don't have control signals. The test plan will also be similar, except it doesn't have the tests for control signals.
7. A 64-bit register for IR. It has one input and four outputs. The test will simply check whether it gives correct four outputs based on the input.
8. The multiplexers. PCSrc, WriteMemDst, WriteDataSrc, ALUSrcA, and ALUSrcB are 2-bit mux. ReadSrc is 1-bit mux. The testing is the same as what we did in class.
9. At this point, we have all the necessary components for assembling. They should all be working correctly since we have tested each of them. Now it's time for assembling.
10. Connect the PC register to PCAdder, and wire the result of PCAdder back to PC register through PCSrc mux. Check whether the PC updates to the result of PCAdder.
11. Connect the PC register to instruction memory, and instruction register. Check whether the instruction register gives four correct outputs.
12. Wire the four outputs of IR to the associated muxes, which connect to the data memory, which is connected to A and B. Check whether A and B store the correct values.
13. Connect A and B to the ALU through the muxes, and connect ALUOut to ALU. Check whether ALUOut has the correct result.
14. Add necessary wires to connect all the registers and muxes. At this point, the datapath should be fully implemented. Test by entering instruction in the instruction memory and check whether PC updates, and whether ALUOut, Ra, and Sp have correct results.
15. Implement the control unit. Test it by checking whether it has the correct output based on the opcode.
16. Wire the output of the control unit to the corresponding control signals and mux controls.
17. Test the full datapath with control unit.

Strategy for testing

We divided the architecture into small components. For each component, we'll write comprehensive tests for it. We'll initialize the inputs, and check whether the actual outputs match the expected outputs.

Errors during testing

1. The PC Adder always has an undefined value at the last bit. It is because we used the built-in Adder16, which requires a carry-in as an input. We forgot to include this carry-in in our tests. We wrote our own PC Adder so that it does not require carry-in.

System Test Plan

1. First, enter the instruction to the .coe file. The instruction is in machine code. We test whether it can read the instruction
2. Then, we enter some test data in the .coe file for data memory, we check whether this data is accessible.
3. Then, we'll enter the control signals manually, to see whether it can update the values in the registers. Also, we should check whether ALU can read the data in the next cycle.
4. We'll check each of our instructions using this method.
5. Finally, we'll enable the control unit to see whether it still works.

I/O Specifications

We don't have I/O registers in our design. However, we do have a reserved memory address to take the input and put the output. In the data memory, the address of 0x0001 and 0x0002 will be the input data. The address of 0x0007 will be the output data.

Performance of Design

1. Our processor implemented 23 instructions to do relPrime and gcd functions. Since each instruction is 8 byte long, it requires **184** bytes to store the Euclid's algorithm and relPrime.
2. Total number of instructions executed when input is 0x13B0: **40842**
3. Total number of cycles: **163420**
4. Average cycles per instruction: **4**
5. Cycle time: **16.890ns = 59.207MHz**
6. The total execution time: **2760163.8ns**
7. Screenshot of performance

```
Device utilization summary:
---------------------------

Selected Device : 3s500efg320-4

 Number of Slices:                      277  out of   4656     5%
 Number of Slice Flip Flops:            190  out of   9312     2%
 Number of 4 input LUTs:                442  out of   9312     4%
 Number of IOs:                           3
 Number of bonded IOBs:                   3  out of    232     1%
 Number of BRAMs:                         5  out of     20    25%
 Number of GCLKs:                         2  out of     24     8%

---------------------------
Partition Resource Summary:
---------------------------

  No Partitions were found in this design.

---------------------------
```

Assembler

We used python to translate the assembly code into hexadecimal instructions. It will first parse the input assembly code. This parser is very comprehensive so that it can parse the assembly code as long as its content is correct. The only rule for the input format is that each instruction takes a whole line. Other than that, the format does not really matter. Then we used the parsed instruction to translate into hexadecimal, and write it to an output text file.

Conclusion

Now we have this fully-implemented Memory to Memory architecture processor. The performance of it is actually better than we expected. Since it is memory to memory, we thought its speed should be slow, However, it finished the algorithm of RelPrime in a short number of cycles (about 160 thousand) and each cycle does not take a long time (only 16ns). It ends up with a CPI of 4. It is larger than other architectures. This is expected because everything depends on the memory, and instructions can only be executed after retrieving the data from the memory. Therefore, most instructions have to wait for the data to be retrieved from the memory. One of the best features we have is that the assembly code is very short in size. It just takes 23 instructions to run the RelPrime algorithm, while the C code takes 21 instructions. It has almost the same length as the C language. However, one of the shortcomings is that each instruction has 64 bits, even if many digits are not used for half of the instructions. This is a waste, but it has to since the slowest instruction needs all these digits.