Faculty of Engineering

Cairo University

CMPS301

Project Report

Ali Ashraf Ali	1210018
Abdelrahman Mohamed Hamdi	1210353
Mohamed Ammar	1210356
Seif Tamer Heakal	1210386

Instruction format of your design

Opcode of each instruction:

Instruction	Opcode
NOP	00000
NOT	00001
NEG	00010
INC	00011
DEC	00100
OUT	00101
IN	00110
MOV	00111
SWAP	01000
ADD	01001
ADDI	01010
SUB	01011
SUBI	01100
AND	01101
OR	01110
XOR	01111
CMP	10000
PUSH	10001
POP	10010

LDM	10011
LDD	10100
STD	10101
PROTECT	10110
FREE	10111
JZ	11000
JMP	11001
CALL	11010
RET	11011
RTI	11100
RESET	11101
INTERRUPT	11110

Instruction bits details:

Field	Bits	Description
Opcode	0-4	Operation code
Source 1	5-7	First source register
Source 2	8-10	Second source register
Destination	11-13	Destination register
Reserved	14-15	Reserved bits (size)
Immediate	16-31	Immediate value or extension

Control Signal Table

Instruction	W/E	M/W	M/R	Logic	M/REG	Sign Ex	ALU pass	Immediate	BRCH
NOP									
NOT				✓					
NEG	✓								
INC	✓								
DEC	✓								
OUT									
IN	✓								
MOV	✓						✓		
SWAP	✓						✓		
ADD	✓								
ADDI	✓							✓	
SUB	✓								
SUBI	✓							✓	
AND	✓			✓					
OR	✓			✓					
XOR	✓			✓					
CMP									
PUSH		✓							

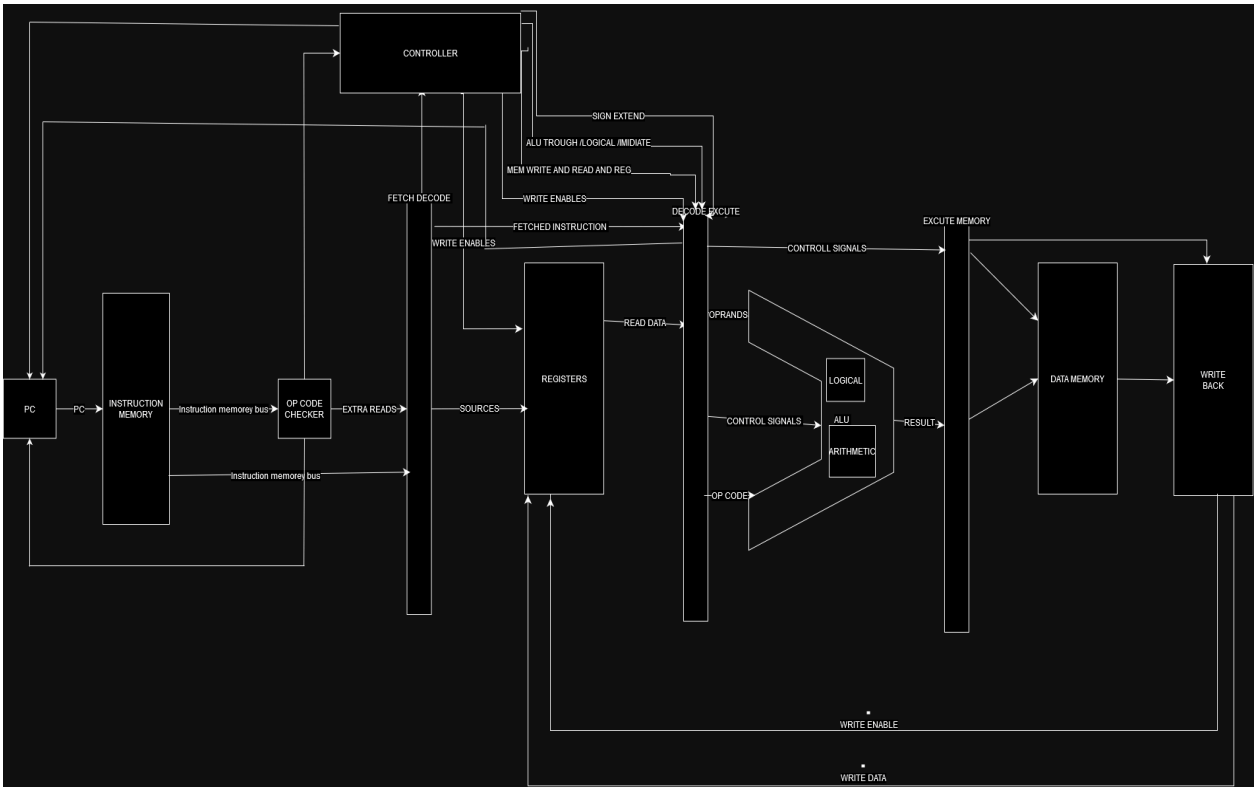
POP	✓		✓		✓				
LDM	✓							✓	
LDD	✓		✓		✓	✓			
STD		✓				✓			
PROTECT									
FREE									
JZ									✓
JMP									✓
CALL		✓							✓
RET			✓						✓
RTI			✓						✓
RESET									
INTERRUPT		✓							

W/E: Write Enable M/W: Memory Write M/R: Memory Read

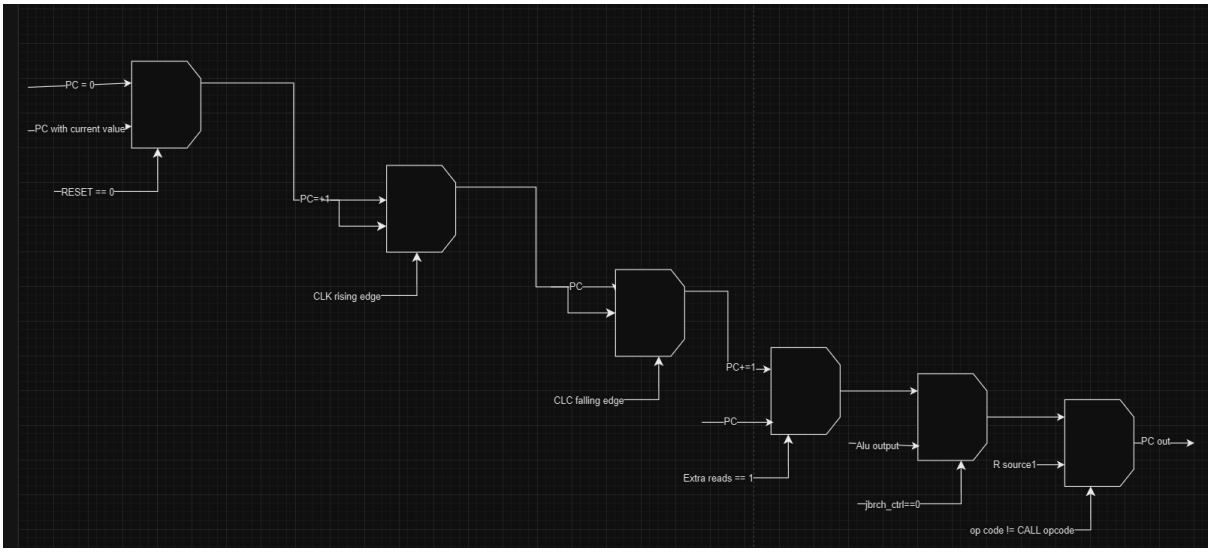
M/Reg: Memory to Register BRCH: Branch Enable

Schematic diagram of the processor with data flow details:

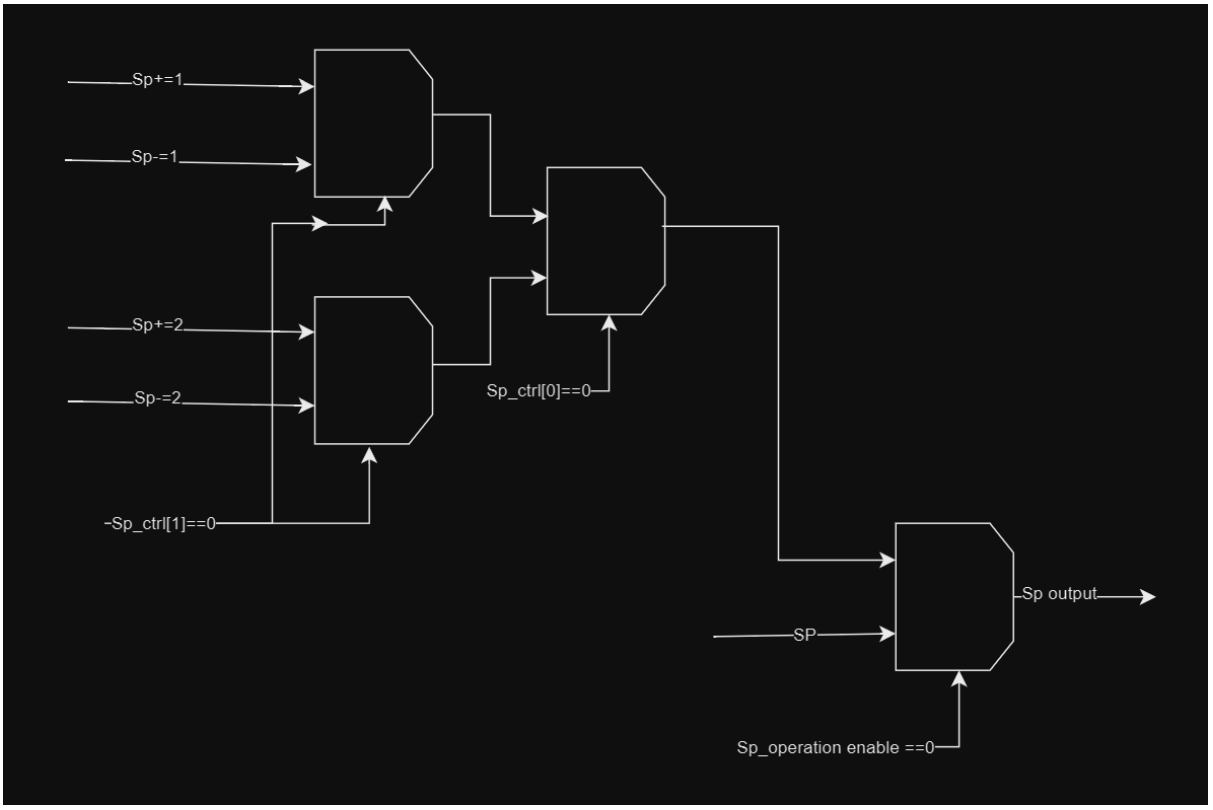
Pipeline processor:



PC increment circuit:



Stack pointer increment circuit:



Pipeline stages design:

1. We have the program counter which keeps track of the memory address of the currently executing instruction and it increments automatically after each instruction fetch, advancing to the next instruction in memory whether by incrementing 1 or 2 based on if the instruction needs extra reads based on the extra reads signals coming from the opcode checker.
2. The instruction memory that contains all the instructions code and have the data memory bus that outputs the instruction
3. We have the op code checker that take a certain bit that we reserved for instructions that will need 32 bits not 6 so that it can set the extra reads signal so we know we will need to increment the pc twice and read on the rising and falling edge so that at the end of the cycle we have the full instruction not just half.
4. Fetch decode register connects the opcode to the register file and the instruction memory also it takes the instruction from the instruction memory and the extra read signals and it is the component that performs taking two cycles from the instruction memory and send to the register file the two sources.
5. The Register file has 2 writes and 2 reads, it takes the write address and data from write back and the write enable from control signals. it reads the first and second sources from the fetched instruction as read addresses
6. The control unit reads the opcode and the bit that tells if it uses an immediate value or not and generates control signals for the alu, memory, register file and the write back

7. The Decode Execute register operates on the falling edge and takes read data from register file and control signals whether it needs to sign extend or not based on the fetched instruction and sends it to the to alu and execute memory register
8. the ALU then takes 2 operands that were decoded in the decode execute register and the sign extended immediate value and the instruction opcode it also takes 4 control signals which are pass through to determine whether the alu will work or just pass the value through, use logic which determines whether its a logical or arithmetic operation if alu_use_logical signal==1 then logical else arithmetic, use immediate which determines whether or not it will use the immediate value and the alu update the flags which but after checking the alu_update_flags signal == 1 it will update
9. the execute memory register takes the write address, write enable, memory write address, memory write, memory read and memory to register from the decode execute register and forwards them to the memory or memory write back
10. The memory works on falling edge and it gets the address its working on from the alu result and it gets an enable for read and write from the execute memory register and it also gets the data in or data out from the execute memory register and output to the write back or using the full forward it will put them to the alu directly.
11. The write back depends on write enable and memory to register it only outputs if write enable is true and if memory to register is true it sends the memory data to register file if its false it sends the alu result to the register file instead.

Pipeline registers details:

Fetch / Decode Register:

Input:

1. CLK
2. Reset
3. raw_instruction
4. extra_reads

Output:

1. out_instruction

Connections:

- raw_instruction: taken from instruction memory through the memory bus
- extra_reads: taken from OP code Checker
- out_instruction: 0 to 4 go to control unit, 5 to 7 go to read address 1 as src1, 8 to 10 go to read address 2 as src2, 11 to 13 are sent to decode execute as write address, bit 14 is sent to control unit to choose if you use immediate value, 14 to 31 are sent to decode execute as immediate instruction

Size: 51 bits

Decode / Execute Register:

Input:

1. CLK
2. write_address
3. write_enable
4. read_data_1
5. read_data_2
6. mem_write
7. mem_read
8. mem_to_reg
9. alu_pass_through
10. alu_use_logical
11. alu_use_immediate
12. alu_update_flags
13. sign_extend_immediate
14. instr_opcode
15. brch_ctrl

Output:

1. out_write_address
2. out_write_enable
3. out_read_data_1
4. out_read_data_2
5. out_mem_write
6. out_mem_read
7. out_mem_to_reg
8. out_alu_pass_through
9. out_alu_use_logical
10. out_alu_use_immediate
11. out_alu_update_flags

12. out_instr_opcode
13. out_instr_immediate
14. out_brch_ctrl

Connections:

- write_address: From fetch decode register
- write_enable: From control component containing control signal
- read_data_1:From Register file to resemble source 1
- read_data_2:From Register file to resemble source 2
- mem_write: From control component containing control signal
- mem_read: From control component containing control signal
- mem_to_reg: From control component containing control signal
- alu_pass_through:From control component that means we pass the source as is without changing anything
- alu_use_logical: From control component that means we will do a logical operation else we will do an arithmetic operation
- alu_use_immediate: From control component that means we have an immediate value that will be used
- alu_update_flags: From control component to signal an enable to update the flags
- sign_extend_immediate: From control component
- instr_opcode: From Fetch decode register
- instr_immediate: From Fetch decode register
- out_instr_immediate: contains the immediate value with the extended bit

- out_alu_pass_through: goes to ALU and it represents whether it will perform an operation or just pass the value
- out_alu_use_logical: goes to ALU and it represents whether it will perform a logical operation or an arithmetic operation
- out_alu_update_flags: goes to ALU and it decides whether or not it will update flags
- out_instr_opcode: goes to ALU and contains the instruction op code
- out_alu_use_immediate: goes to ALU and decides whether to use immediate value

size: 115 bits

Execute / Memory register:

Input:

1. CLK
2. write_address
3. write_enable
4. mem_write_data
5. mem_write
6. mem_read
7. mem_to_reg
8. alu_result

output:

1. out_write_address

2. out_write_enable
3. out_mem_write_data
4. out_mem_write
5. out_mem_read
6. out_mem_to_reg
7. out_alu_result

Connections:

- write_address: From Decode execute register write address
- write_enable: From Decode execute register write enable containing control signal
- mem_write_data: From Decode Execute operand 2 because data are there
- mem_write: From Decode execute register containing control signal
- mem_read: From Decode execute register containing control signal
- mem_to_reg: From Decode execute register containing control signal
- alu_result: From Alu containing the result

size: 72 bits

<u>Write Back Register:</u>

input:

1. CLK
2. write_enable
3. write_address
4. alu_result

5. mem_data
6. mem_to_reg

output:

1. out_write_enable
2. out_write_address
3. out_write_data

Connections:

- write_enable: From the Execute Memory register containing control signal
- write_address: From the Execute Memory register containing the write address
- alu_result: From the Execute Memory register containing the alu result
- mem_data: From the Data Memory Component containing the output from the Data memory data
- mem_to_reg: From the Execute Memory register to choose from whether to use data from the data memory or the alu result in the write back

size: 70

Pipeline hazards:

Structural hazard:

I can not read and write from the register file at the same time so maybe i will have a write back and a register write from the fetched instructions at the same time so i will need to insert a nop operation if the register file is being written into by the fetch decode and the write back at the same time.

data hazards:

If i want to use a register that will be updated with the write back such LDm it will still not be updated so we can use full data forwarding to send the data from the memory to the next alu immediately not have to wait but if i use the LDD and use the same register i will have to use nop operations or use full forward and of course full forwarding is better to save some cycles and not to be wasteful

branching hazards: i will need to branch using the JMP or JZ or call or RET or RTI all of this will need to flush or through out the unnecessary instructions but we can try to utilise the global bit methode to always assume we don't jump at first and if i was right i will save three cycles compared to stall as i will know if i am a jump or not at the alu to save hardware because we already calculate the address there and if i was wrong just once i will switch to assume not taken and if i was wrong i will change again // explain in details :: we'll have a single bit state that represents the prediction for all branches in the program. This single bit state will represent the prediction for all branches It indicates whether the next branch encountered is likely to be taken or not based on the history of all branches . When a branch instruction is encountered, the processor uses the global predictor state to predict whether the branch will be

taken or not. If the prediction matches the actual outcome, it's considered correct. If not, the predictor state is updated based on the actual outcome of the branch.