Project

General Description

Project Overview

"Processor design is the design engineering task of creating a processor, a key component of computer hardware. The design process involves choosing an instruction set and a certain execution paradigm, and results in a microarchitecture. The mode of operation of any processor is the execution of lists of instructions. Instructions typically include those to compute or manipulate data values using registers, change or retrieve values in read/write memory, perform relational tests between data values and to control program flow."

In this project, you will simulate a fictional processor design and architecture using C. You are asked to choose one of four processor packages described in the upcoming sections.

You are required to read the packages details, the general important notes, the deliverables and instructions CAREFULLY.

Project

General Description

Detailed Description

Packages Description

Package 1: Spicy Von Neumann Fillet with extra shifts

Memory Architecture

a) Architecture: Von Neumann

Von Neumann Architecture is a digital computer architecture whose design is based on the concept of stored program computers where program data and instruction data are stored in the same memory.

b) **Memory Size**: 2048 * 32

Main Memory	
2048 Rows	32 Bits / Row
	Data (1024 to 2047)
	Instructions (0 to 1023)

- The main memory addresses are from 0 to (2^11) - 1 (0 to 2047).
- Each memory block (row) contains 1 word which is 32 bits (4 bytes).
- The main memory is word addressable.
- Addresses from 0 to 1023 contain the program instructions.
- Addresses from 1024 to 2048 contain the data.

c) **Registers**: 33

• Size: 32 bits
• 31 General-Purpose Registers (GPRS)
– Names: R1 to R31
• 1 Zero Register
– Name: R0
– Hard-wired value “0” (cannot be overwritten by any instruction).
• 1 Program Counter
– Name: PC
– A program counter is a register in a computer processor that contains the address (location) of the instruction being executed at the current time.

– As each instruction gets fetched, the program counter is incremented to point to the next instruction to be executed.

Instruction Set Architecture

a) Instruction Size: 32 bits
b) Instruction Types: 3

R-Format				
OPCODE	**R1**	**R2**	**R3**	**SHAMT**
4	5	5	5	13

I-Format			
OPCODE	**R1**	**R2**	**IMMEDIATE**
4	5	5	18

J-Format	
OPCODE	**ADDRESS**
4	28

c) Instruction Count: 12

• The opcodes are from 0 to 11 according to the instructions order in the following table:

Name	Mnemonic	Type	Format	Operation
Add	ADD	R	ADD R1 R2 R3	R1 = R2 + R3
Subtract	SUB	R	SUB R1 R2 R3	R1 = R2 - R3
Multiply Immediate	MULI	I	MULI R1 R2 IMM	R1 = R2 * IMM
Add Immediate	ADDI	I	ADDI R1 R2 IMM	R1 = R2 + IMM
Branch if Not Equal	BNE	I	BNE R1 R2 IMM	IF(R1 != R2) { PC = PC+1+IMM }
And Immediate	ANDI	I	ANDI R1 R2 IMM	R1 = R2 & IMM
Or Immediate	ORI	I	ORI R1 R2 IMM	R1 = R2 | IMM
Jump	J	J	J ADDRESS	PC = PC[31:28] || ADDRESS
Shift Left Logical*	SLL	R	SLL R1 R2 SHAMT	R1 = R2 << SHAMT
Shift Right Logical*	SRL	R	SRL R1 R2 SHAMT	R1 = R2 >>> SHAMT
Load Word	LW	I	LW R1 R2 IMM	R1 = MEM[R2 + IMM]
Store Word	SW	I	SW R1 R2 IMM	MEM[R2 + IMM] = R1

∗ SLL and SRL: R3 will be 0 in the instruction format.
"||" symbol indicates concatenation (0100 || 1100 = 01001100)

Data Path

a) Stages: 5

• All instructions regardless of their type must pass through all 5 stages even if they do not need to access a particular stage.
• **Instruction Fetch (IF):** Fetches the next instruction from the main memory using the address in the PC (Program Counter), and increments the PC.
• **Instruction Decode (ID):** Decodes the instruction and reads any operands required from the register file.
• **Execute (EX):** Executes the instruction. In fact, all ALU operations are done in this stage.
• **Memory (MEM):** Performs any memory access required by the current instruction. For loads, it would load an operand from the main memory, while for stores, it would store an operand into the main memory.
• **Write Back (WB):** For instructions that have a result (a destination register), the Write Back writes this result back to the register file.

b) Pipeline: 4 instructions (maximum) running in parallel

• **Instruction Fetch (IF)** and **Memory (MEM)** can not be done in parallel since they access the same physical memory.
• At a given clock cycle, you can either have the **IF**, **ID**, **EX**, **WB** stages active, or the **ID**, **EX**, **MEM**, **WB** stages active.
• Number of clock cycles: $7 + ((n - 1) * 2)$, where n = number of instructions
– Imagine a program with 7 instructions:
$* 7 + (6 * 2) = 19$ clock cycles
– You are required to understand the pattern in the example and implement it.

Project

General Description

Package 1 Pipeline					
	Instruction Fetch (IF)	**Instruction Decode (ID)**	**Execute (EX)**	**Memory (MEM)**	**Write Back (WB)**
Cycle 1	Instruction 1				
Cycle 2		Instruction 1			
Cycle 3	Instruction 2	Instruction 1			
Cycle 4		Instruction 2	Instruction 1		
Cycle 5	Instruction 3	Instruction 2	Instruction 1		
Cycle 6		Instruction 3	Instruction 2	Instruction 1	
Cycle 7	Instruction 4	Instruction 3	Instruction 2		Instruction 1
Cycle 8		Instruction 4	Instruction 3	Instruction 2	
Cycle 9	Instruction 5	Instruction 4	Instruction 3		Instruction 2
Cycle 10		Instruction 5	Instruction 4	Instruction 3	
Cycle 11	Instruction 6	Instruction 5	Instruction 4		Instruction 3
Cycle 12		Instruction 6	Instruction 5	Instruction 4	
Cycle 13	Instruction 7	Instruction 6	Instruction 5		Instruction 4
Cycle 14		Instruction 7	Instruction 6	Instruction 5	
Cycle 15		Instruction 7	Instruction 6		Instruction 5
Cycle 16			Instruction 7	Instruction 6	
Cycle 17			Instruction 7		Instruction 6
Cycle 18				Instruction 7	
Cycle 19					Instruction 7

• The pattern is as follows:
– You fetch an instruction every 2 clock cycles starting from clock cycle 1.
– An instruction stays in the Decode (ID) stage for 2 clock cycles.
– An instruction stays in the Execute (EX) stage for 2 clock cycles.
– An instruction stays in the Memory (MEM) stage for 1 clock cycle.
– An instruction stays in the Write Back (WB) stage for 1 clock cycle.
– You can not have the Instruction Fetch (IF) and Memory (MEM) stages working in parallel. Only one of them is active at a given clock cycle.

Project

General Description

Package 2: Fillet-O-Neumann with moves on the side

Memory Architecture

a) Architecture: Von Neumann

• Von Neumann Architecture is a digital computer architecture whose design is based on the concept of stored program computers where program data and instruction data are stored in the same memory.

b) Memory Size: 2048 * 32

Main Memory	
2048 Rows	32 Bits / Row
	Data (1024 to 2047)
	Instructions (0 to 1023)

• The main memory addresses are from 0 to 2^11 − 1 (0 to 2047).
• Each memory block (row) contains 1 word which is 32 bits (4 bytes).
• The main memory is word addressable.
• Addresses from 0 to 1023 contain the program instructions.
• Addresses from 1024 to 2048 contain the data.

c) Registers: 33

• Size: 32 bits
• 31 General-Purpose Registers (GPRS)
– Names: R1 to R31
• 1 Zero Register
– Name: R0
– Hard-wired value “0” (cannot be overwritten by any instruction).
• 1 Program Counter
– Name: PC
– A program counter is a register in a computer processor that contains the address(location) of the instruction being executed at the current time.
– As each instruction gets fetched, the program counter is incremented to point to the next instruction to be executed.

Instruction Set Architecture

a) Instruction Size: 32 bits
b) Instruction Types: 3

R-Format				
OPCODE	R1	R2	R3	SHAMT
4	5	5	5	13

I-Format			
OPCODE	R1	R2	IMMEDIATE
4	5	5	18

J-Format	
OPCODE	ADDRESS
4	28

c) Instruction Count: 12

• The opcodes are from 0 to 11 according to the instructions order in the following table:

Name	Mnemonic	Type	Format	Operation
Add	ADD	R	ADD R1 R2 R3	R1 = R2 + R3
Subtract	SUB	R	SUB R1 R2 R3	R1 = R2 - R3
Multiply	MUL	R	MUL R1 R2 R3	R1 = R2 * R3
Move Immediate*	MOVI	I	MOVI R1 IMM	R1 = IMM
Jump if Equal	JEQ	I	JEQ R1 R2 IMM	IF(R1 == R2) { PC = PC+1+IMM }
And	AND	R	AND R1 R2 R3	R1 = R2 & R3
Exclusive Or Immediate	XORI	I	XORI R1 R2 IMM	R1 = R2 ⊕ IMM
Jump	JMP	J	JMP ADDRESS	PC = PC[31:28] || ADDRESS
Logical Shift Left**	LSL	R	LSL R1 R2 SHAMT	R1 = R2 << SHAMT
Logical Shift Right**	LSR	R	LSR R1 R2 SHAMT	R1 = R2 >>> SHAMT
Move to Register	MOVR	I	MOVR R1 R2 IMM	R1 = MEM[R2 + IMM]
Move to Memory	MOVM	I	MOVM R1 R2 IMM	MEM[R2 + IMM] = R1

* MOVI: R2 will be 0 in the instruction format.
** LSL and LSR: R3 will be 0 in the instruction format.
"||" symbol indicates concatenation (0100 || 1100 = 01001100).

Data Path

a) Stages: 5

• All instructions regardless of their type must pass through all 5 stages even if they do not need to access a particular stage.

Project

General Description

• **Instruction Fetch (IF):** Fetches the next instruction from the main memory using the address in the PC (Program Counter), and increments the PC.
• **Instruction Decode (ID):** Decodes the instruction and reads any operands required from the register file.
• **Execute (EX):** Executes the instruction. In fact, all ALU operations are done in this stage.
• **Memory (MEM):** Performs any memory access required by the current instruction. For loads, it would load an operand from the main memory, while for stores, it would store an operand into the main memory.
• **Write Back (WB):** For instructions that have a result (a destination register), the Write Back writes this result back to the register file.

b) Pipeline: 4 instructions (maximum) running in parallel

• **Instruction Fetch (IF)** and **Memory (MEM)** can not be done in parallel since they access the same physical memory.
• At a given clock cycle, you can either have the **IF**, **ID**, **EX**, **WB** stages active, or the **ID**, **EX**, **MEM**, **WB** stages active.
• Number of clock cycles: $7 + ((n - 1) * 2)$, where n = number of instructions
– Imagine a program with 7 instructions: $7 + (6 * 2) = 19$ clock cycles
– You are required to understand the pattern in the example and implement it.

Project

General Description

Package 3 Pipeline					
	Instruction Fetch (IF)	**Instruction Decode (ID)**	**Execute (EX)**	**Memory (MEM)**	**Write Back (WB)**
Cycle 1	Instruction 1				
Cycle 2		Instruction 1			
Cycle 3	Instruction 2	Instruction 1			
Cycle 4		Instruction 2	Instruction 1		
Cycle 5	Instruction 3	Instruction 2	Instruction 1		
Cycle 6		Instruction 3	Instruction 2	Instruction 1	
Cycle 7	Instruction 4	Instruction 3	Instruction 2		Instruction 1
Cycle 8		Instruction 4	Instruction 3	Instruction 2	
Cycle 9	Instruction 5	Instruction 4	Instruction 3		Instruction 2
Cycle 10		Instruction 5	Instruction 4	Instruction 3	
Cycle 11	Instruction 6	Instruction 5	Instruction 4		Instruction 3
Cycle 12		Instruction 6	Instruction 5	Instruction 4	
Cycle 13	Instruction 7	Instruction 6	Instruction 5		Instruction 4
Cycle 14		Instruction 7	Instruction 6	Instruction 5	
Cycle 15		Instruction 7	Instruction 6		Instruction 5
Cycle 16			Instruction 7	Instruction 6	
Cycle 17			Instruction 7		Instruction 6
Cycle 18				Instruction 7	
Cycle 19					Instruction 7

• The pattern is as follows:
– You fetch an instruction every 2 clock cycles starting from clock cycle 1.
– An instruction stays in the Decode (ID) stage for 2 clock cycles.
– An instruction stays in the Execute (EX) stage for 2 clock cycles.
– An instruction stays in the Memory (MEM) stage for 1 clock cycle.
– An instruction stays in the Write Back (WB) stage for 1 clock cycle.
– You can not have the Instruction Fetch (IF) and Memory (MEM) stages working in parallel. Only one of them is active at a given clock cycle.

Project

General Description

Package 3: Double Big Harvard combo large arithmetic shifts

Memory Architecture

a) Architecture: Harvard
   • Harvard Architecture is the digital computer architecture whose design is based on the concept where there are separate storage and separate buses (signal path) for instruction and data. It was basically developed to overcome the bottleneck of Von Neumann Architecture.

b) Instruction Memory Size: 1024 * 16

Instruction Memory	
	16 Bits / Row
1024 Rows	

   • The instruction memory addresses are from 0 to 2^10 − 1 (0 to 1023).
   • Each memory block (row) contains 1 word which is 16 bits (2 bytes).
   • The instruction memory is word addressable.
   • The program instructions are stored in the instruction memory

c) Data Memory Size: 2048 * 8

Data Memory	
	8 Bits / Row
2048 Rows	

• The data memory addresses are from 0 to 2^11 − 1 (0 to 2047).
• Each memory block (row) contains 1 word which is 8 bits (1 byte).
• The data memory is word/byte addressable (1 word = 1 byte).
• The data is stored in the data memory.

d) Registers: 66
   • Size: 8 bits
   • 64 General-Purpose Registers (GPRS)

General Description

– Names: R0 to R63

• 1 Status Register

7	6	5	4	3	2	1	0
0	0	0	C	V	N	S	Z

– Name: SREG

– A status register, flag register, or condition code register (CCR) is a collection of status flag bits for a processor.

– The status register has 5 flags updated after the execution of specific instructions:

∗ Carry Flag (C): Indicates when an arithmetic carry or borrow has been generated out of the most significant bit position.

· Check on 9th bit (bit 8) of UNSIGNED[VALUE1] OP UNSIGNED[VALUE2] == 1 or not.

∗ Two's Complement Overflow Flag (V): Indicates when the result of a signed number operation is too large, causing the high-order bit to overflow into the sign bit.

· If 2 numbers are added, and they both have the same sign (both positive or both negative), then overflow occurs (V = 1) if and only if the result has the opposite sign. Overflow never occurs when adding operands with different signs.

· If 2 numbers are subtracted, and their signs are different, then overflow occurs (V = 1) if and only if the result has the same sign as the subtrahend.

∗ Negative Flag (N): Indicates a negative result in an arithmetic or logic operation.

· N = 1 if result is negative.

· N = 0 if result is positive or zero.

∗ Sign Flag (S): Indicates the expected sign of the result (not the actual sign).

· S = N ⊕ V (XORing the negative and overflow flags will calculate the sign flag).

∗ Zero Flag (Z): Indicates that the result of an arithmetic or logical operation was zero.

· Z = 1 if result is 0.

· Z = 0 if result is not 0.

∗ Since all registers are 8 bits, and we are only using 5 bits in the Status Register for the flags, you are required to keep Bits7:5 cleared "0" at all times in the register.

• 1 Program Counter

– Name: PC

– Type: Special-purpose register with a size of 16 bits (not 8 bits).

– A program counter is a register in a computer processor that contains the address(location) of the instruction being executed at the current time.

– As each instruction gets fetched, the program counter is incremented to point to the next instruction to be executed.

Project

General Description

Instruction Set Architecture

a) Instruction Size: 16 bits
b) Instruction Types: 2

R-Format		
OPCODE	**R1**	**R2**
4	6	6

I-Format		
OPCODE	**R1**	**IMMEDIATE**
4	6	6

c) Instruction Count: 12
• The opcodes are from 0 to 11 according to the instructions order in the following table:

Name	Mnemonic	Type	Format	Operation
Add	ADD	R	ADD R1 R2	R1 = R1 + R2
Subtract	SUB	R	SUB R1 R2	R1 = R1 - R2
Multiply	MUL	R	MUL R1 R2	R1 = R1 * R2
Move Immediate	MOVI	I	MOVI R1 IMM	R1 = IMM
Branch if Equal Zero	BEQZ	I	BEQZ R1 IMM	IF(R1 == 0) { PC = PC+1+IMM }
And Immediate	ANDI	I	ANDI R1 IMM	R1 = R1 & IMM
Exclusive Or	EOR	R	EOR R1 R2	R1 = R1 ⊕ R2
Branch Register	BR	R	BR R1 R2	PC = R1 || R2
Shift Arithmetic Left	SAL	I	SAL R1 IMM	R1 = R1 << IMM
Shift Arithmetic Right	SAR	I	SAR R1 IMM	R1 = R1 >> IMM
Load to Register	LDR	I	LDR R1 ADDRESS	R1 = MEM[ADDRESS]
Store from Register	STR	I	STR R1 ADDRESS	MEM[ADDRESS] = R1

"||" symbol indicates concatenation (0100 || 1100 = 01001100).

d) The Status Register (SREG) flags are affected by the following instructions:
• The Carry flag (C) is updated every ADD instruction.
• The Overflow flag (V) is updated every ADD and SUB instruction.
• The Negative flag (N) is updated every ADD, SUB, MUL, ANDI, EOR, SAL, and SAR instruction.
• The Sign flag (S) is updated every ADD and SUB instruction.
• The Zero flag (Z) is updated every ADD, SUB, MUL, ANDI, EOR, SAL, and SAR instruction.
• A flag value can only be updated by the instructions related to it.

Data Path

a) Stages: 3

• All instructions regardless of their type must pass through all 3 stages.
• **Instruction Fetch (IF):** Fetches the next instruction from the main memory using the address in the PC (Program Counter), and increments the PC.
• **Instruction Decode (ID):** Decodes the instruction and reads any operands required from the register file.
• **Execute (EX):** Executes the instruction. In fact, all ALU operations are done in this stage. Moreover, it performs any memory access required by the current instruction. For loads, it would load an operand from the main memory, while for stores, it would store an operand into the main memory. Finally, for instructions that have a result (a destination register), it writes this result back to the register file.

b) **Pipeline: 3** instructions (maximum) running in parallel
• Number of clock cycles: $3 + ((n - 1) * 1)$, where n = number of instructions
– Imagine a program with 7 instructions:
$* 3 + (6 * 1) = 9$ clock cycles
– You are required to understand the pattern in the example and implement it.

Package 4 Pipeline			
	Instruction Fetch (IF)	**Instruction Decode (ID)**	**Execute (EX)**
Cycle 1	Instruction 1		
Cycle 2	Instruction 2	Instruction 1	
Cycle 3	Instruction 3	Instruction 2	Instruction 1
Cycle 4	Instruction 4	Instruction 3	Instruction 2
Cycle 5	Instruction 5	Instruction 4	Instruction 3
Cycle 6	Instruction 6	Instruction 5	Instruction 4
Cycle 7	Instruction 7	Instruction 6	Instruction 5
Cycle 8		Instruction 7	Instruction 6
Cycle 9			Instruction 7

Project

General Description

Package 4: Double McHarvard with cheese circular shifts

Memory Architecture

a) Architecture: Harvard

• Harvard Architecture is the digital computer architecture whose design is based on the concept where there are separate storage and separate buses (signal path) for instruction and data. It was basically developed to overcome the bottleneck of Von Neumann Architecture.

b) Instruction Memory Size: 1024 * 16

Instruction Memory	
	16 Bits / Row
1024 Rows	

- The instruction memory addresses are from 0 to 2^10 − 1 (0 to 1023).
- Each memory block (row) contains 1 word which is 16 bits (2 bytes).
- The instruction memory is word addressable.
- The program instructions are stored in the instruction memory

c) Data Memory Size: 2048 * 8

Data Memory	
	8 Bits / Row
2048 Rows	

• The data memory addresses are from 0 to 2^11 − 1 (0 to 2047).
• Each memory block (row) contains 1 word which is 8 bits (1 byte).
• The data memory is word/byte addressable (1 word = 1 byte).
• The data is stored in the data memory.

d) Registers: 66

• Size: 8 bits
• 64 General-Purpose Registers (GPRS)
– Names: R0 to R63
• 1 Status Register

7	6	5	4	3	2	1	0
0	0	0	C	V	N	S	Z

– Name: SREG
– A status register, flag register, or condition code register (CCR) is a collection of status flag bits for a processor.
– The status register has 5 flags updated after the execution of specific instructions:

∗ Carry Flag (C): Indicates when an arithmetic carry or borrow has been generated out of the most significant bit position.
· Check on 9th bit (bit 8) of UNSIGNED[VALUE1] OP UNSIGNED[VALUE2] ==1 or not.
∗ Two's Complement Overflow Flag (V): Indicates when the result of a signed number operation is too large, causing the high-order bit to overflow into the sign bit.

· If 2 numbers are added, and they both have the same sign (both positive or both negative), then overflow occurs (V = 1) if and only if the result has the opposite sign. Overflow never occurs when adding operands with different signs.

· If 2 numbers are subtracted, and their signs are different, then overflow occurs (V=1) if and only if the result has the same sign as the subtrahend.
∗ Negative Flag (N): Indicates a negative result in an arithmetic or logic operation.
· N = 1 if result is negative.
· N = 0 if result is positive or zero.
∗ Sign Flag (S): Indicates the expected sign of the result (not the actual sign).
· S = N ⊕ V (XORing the negative and overflow flags will calculate the sign flag).
∗ Zero Flag (Z): Indicates that the result of an arithmetic or logical operation was zero.
· Z = 1 if result is 0.
· Z = 0 if result is not 0.

∗ Since all registers are 8 bits, and we are only using 5 bits in the Status Register for the flags, you are required to keep Bits7:5 cleared "0" at all times in the register.

• 1 Program Counter

– Name: PC
– Type: Special-purpose register with a size of 16 bits (not 8 bits).
– A program counter is a register in a computer processor that contains the address(location) of the instruction being executed at the current time.
– As each instruction gets fetched, the program counter is incremented to point to the next instruction to be executed.

Instruction Set Architecture

a) Instruction Size: 16 bits

b) Instruction Types: 2

R-Format		
OPCODE	R1	R2
4	6	6

I-Format		
OPCODE	R1	IMMEDIATE
4	6	6

c) Instruction Count: 12
• The opcodes are from 0 to 11 according to the instructions order in the following table:

Project

General Description

Name	Mnemonic	Type	Format	Operation
Add	ADD	R	ADD R1 R2	R1 = R1 + R2
Subtract	SUB	R	SUB R1 R2	R1 = R1 - R2
Multiply	MUL	R	MUL R1 R2	R1 = R1 * R2
Load Immediate	LDI	I	LDI R1 IMM	R1 = IMM
Branch if Equal Zero	BEQZ	I	BEQZ R1 IMM	IF(R1 == 0) { PC = PC+1+IMM }
And	AND	R	AND R1 R2	R1 = R1 & R2
Or	OR	R	OR R1 R2	R1 = R1 | R2
Jump Register	JR	R	JR R1 R2	PC = R1 || R2
Shift Left Circular	SLC	I	SLC R1 IMM	R1 = R1 << IMM | R1 >>> 8 - IMM
Shift Right Circular	SRC	I	SRC R1 IMM	R1 = R1 >>> IMM | R1 << 8 - IMM
Load Byte	LB	I	LB R1 ADDRESS	R1 = MEM[ADDRESS]
Store Byte	SB	I	SB R1 ADDRESS	MEM[ADDRESS] = R1

"||" symbol indicates concatenation (0100 || 1100 = 01001100).

d) The Status Register (SREG) flags are affected by the following instructions:
- The Carry flag (C) is updated every ADD instruction.
- The Overflow flag (V) is updated every ADD and SUB instruction.
- The Negative flag (N) is updated every ADD, SUB, MUL, AND, OR, SLC, and SRC instruction.
- The Sign flag (S) is updated every ADD and SUB instruction.
- The Zero flag (Z) is updated every ADD, SUB, MUL, AND, OR, SLC, and SRC instruction.
- A flag value can only be updated by the instructions related to it.

Data Path

a) Stages: 3
- All instructions regardless of their type must pass through all 3 stages.
- **Instruction Fetch (IF):** Fetches the next instruction from the main memory using the address in the PC (Program Counter), and increments the PC.
- **Instruction Decode (ID):** Decodes the instruction and reads any operands required from the register file.
- **Execute (EX):** Executes the instruction. In fact, all ALU operations are done in this stage. Moreover, it performs any memory access required by the current instruction. For loads, it would load an operand from the main memory, while for stores, it would store an operand into the main memory. Finally, for instructions that have a result (a destination register), it writes this result back to the register file.

b) **Number of clock cycles:** $3 + ((n - 1) * 1)$, where n = number of instructions

Project

General Description

– Imagine a program with 7 instructions:
∗ 3 + (6 ∗ 1) = 9 clock cycles
– You are required to understand the pattern in the example and implement it.

Package 2 Pipeline			
	Instruction Fetch (IF)	**Instruction Decode (ID)**	**Execute (EX)**
Cycle 1	Instruction 1		
Cycle 2	Instruction 2	Instruction 1	
Cycle 3	Instruction 3	Instruction 2	Instruction 1
Cycle 4	Instruction 4	Instruction 3	Instruction 2
Cycle 5	Instruction 5	Instruction 4	Instruction 3
Cycle 6	Instruction 6	Instruction 5	Instruction 4
Cycle 7	Instruction 7	Instruction 6	Instruction 5
Cycle 8		Instruction 7	Instruction 6
Cycle 9			Instruction 7

Project

General Description

Important Notes

Part 1: Quick Remarks

1. The PC starts from 0.
    - The first instruction will be stored at address 0 in all packages.
2. In Packages 1 and 2, R0 (Zero Register) can be the destination in an instruction.
    - The instruction must continue normally in the pipeline.
    - However, it won't overwrite the value of R0.
    - The value will remain 0 in all cases. You must **not** throw an error if R0 is the destination register.
    - Let the instruction continue normally in the pipeline, but don't overwrite the value of R0.
3. For each clock cycle, you need to print which instruction is in each stage, as well as, the values that entered the stage, and the output of this stage.
    - Moreover, if you changed the value of a location in the memory or the register file, you need to print that this location or register (including R0) value has changed alongside the new value (and in which stage did the value change).
    - At the end of your program, you need to print the values of all registers (general and special purpose including the PC and SREG), and the full instruction and data memory locations.
4. Immediate values are signed (2's complement), which means they can be positive or negative. The only exception is the "shift" instructions, where the immediate will always be positive.
5. You are required to make sure that the data types used match the project description.
    - If an instruction is 32 bits, you need to use "Integer" or if you use a String of 0s and 1s, then you must ensure that the string contains 32 characters.
    - If an instruction is 16 bits, you need to use "short int", or a string of 16 characters.
    - If a value is 8 bits, you need to use "char" or “int_8” from the <stdint> library.
6. For Packages 3 and 4, check for an example on the Carry flag calculation at the end of this document.
7. When parsing the text file containing the assembly instructions, you are not allowed to keep track of the fields.
    - You should create a decimal/binary/String of 0s and 1s concatenated instructions to be stored in the instruction memory.
    - Then, during the decode stage, you will decode the instruction into ALL POSSIBLE FORMATS.
    - Any help inserted from the parsing stage will not be correct.

8. In the conditional branch instructions in all packages, "PC + 1" is the PC that was already incremented during the fetch stage.
    - You are not required to increment it again. It just indicates that the PC of the branch instruction will be incremented by 1, which was done during the fetch stage already.
    - Check Part 5 in this post to understand more.
9. You should NOT pre-calculate the total cycles that an assembly program will take and use it as a stopping condition for the simulation. Only use it as a point of reference when you are debugging. The correct way of stopping would be that there are no more instructions to fetch.

Part 2: Branches and Jumps 1.0

1. Clarification regarding the conditional branches/jumps for **ALL PACKAGES**:
    - PC = PC + 1 + IMM = Address of branch instruction + 1 (to point to the next instruction) + immediate
    - Example (using the instructions from Package 1):

```
0: BNE R1 R2 2
1: ADD R1 R2 R3
2: ADD R4 R5 R6
3: ADD R7 R8 R9
if(R1 != R2) {
      PC = address of branch instruction + 1 "to point to next
instruction" + 2 "offset"
}
```

    - **This applies to:**
        - Package 1: Branch if Not Equal (BNE)
        - Package 2: Jump if Equal (JEQ)
        - Package 3: Branch if Equal Zero (BEQZ)
        - Package 4: Branch if Equal Zero (BEQZ)
2. Also, the unconditional jump instructions (J and JMP) in Packages 1 and 2 will have the following format:

```
0: J 2 // JMP 2 // Address = PC[31:28] + 2 = 2 since the most
significant 4 bits of the current PC are 0s.
1: ADD R1 R2 R3
2: ADD R4 R5 R6
3: ADD R7 R8 R9
```

3. While the Jump Register (JR) and Branch Register (BR) instructions in Packages 3 and 4 will have the following format:

```
0: JR R0 R1 // BR R0 R1
1: ADD R1 R2 R3
2: ADD R4 R5 R6
3: ADD R7 R8 R9
// Address = Concatenation between the value of R0 and R1 where R0 is the most significant byte and R1 is the least significant byte
// Address = 00000000 00000000 (since R0 and R1 initially contain the value 0)
```

Part 3: Braches and Jumps 2.0

- Normally, the PC is updated in the **Fetch** stage to point to the next instruction.
  - `PC = PC + 1`
- However, during the Execute stage of a conditional branch/jump instruction, the PC can be updated with the branch address if the condition is true.
- Regarding the unconditional jump instructions, you will also update the PC with the jump address during the **Execute** stage in all packages.
- In all cases, if the condition is true, or we are unconditionally jumping, all instructions that entered the datapath after the branch/jump instruction must be ignored (not executed and dropped) and we should start fetching the instruction we branched/jumped to.
- In Packages 3 and 4, you will fetch the instruction (that we branched/jumped to) directly in the following clock cycle after the Execute stage of the branch/jump instruction.
- In Packages 1 and 2, you must wait for the branch/jump instruction to finish the 2 clock cycles it will spend in the Execute stage, then the branch/jump instruction will move to the Memory stage in the following cycle while nothing will be done in the Fetch stage.
- Thus, we will fetch the instruction (that we branched/jumped to) after 2 cycles after finishing the Execute stage of branch/jump instruction.

```
- X: Branch/Jump in Execute (1st time) + an instruction in the decode stage
- X+1: Branch/Jump in Execute (2nd time) + an instruction in the decode stage + an instruction in the fetch stage
- X+2: Branch/Jump in Memory (nothing will be fetched since we are using the memory in the memory stage) + an instruction in the execute stage + an instruction in the decode stage
- X+3 Fetch new instruction (branched/jumped) + Branch/Jump in Write Back + drop the other 2 instructions in the datapath
```

- Any instruction fetched while we are calculating the branch address and condition or the jump address must be dropped from the datapath when we branch or jump.

Project

General Description

Part 4: Hazards

- You are not required to handle Data Hazards.
- Regarding Control Hazards, you only need to do the following:
- As mentioned before, the PC is updated with the branch/jump address during the execute stage for all packages.
- You need to make sure that during the execute stage if you are updating the PC with the branch or jump address, you need to remove the instructions in the decode and fetch stages (flush the instructions that entered the pipeline after the branch/jump instruction), and in the upcoming clock cycle, you will fetch the instruction pointed to by the new PC.
- However, the branch/jump instruction which was at the execute stage will move to the memory stage (Packages 1 and 2).
- Imagine if we have a branch instruction followed by "add" and "sub" instructions.
- If the branch condition is "taken", which means that we will branch, we will update the PC in the execute stage with the address of the new instruction that we should jump to.
- However, the "add" and "sub" instructions were already in the pipeline, so we need to remove them.

Example:

(It does not represent any of the 4 packages)

```
Cycles 1 2 3 4 5 6 7 8
Branch F D E M W
ADD   F D
SUB         F
NEW         F D E M W
```

Note: NEW refers to the new instruction that we branched/jumped to.

- Also, in Packages 3 and 4, the execute stage is the last stage since we don't have memory and writeback stages.
- However, in Packages 1 and 2, the decode and execute stages take 2 clock cycles each, and we only fetch a new instruction during an odd clock cycle, so you'll update the PC in the second execute cycle, and then wait for 2 cycles to fetch the new instruction.
- The above example applies to all of the unconditional jump instructions (J, JMP, BR, JR).
- Also, it applies to all of the conditional jump instructions (BEQZ, BNE, JEQ) if the condition is true.
- If the condition is false, we will continue executing the "add" and "sub" instructions normally.
- **How to remove/flush an instruction from the pipeline?**
  - You can remove it by clearing all the variables associated with it, or by using flags.

- You must make sure that the removed instructions won't affect the registers' values and the data memory values. Other than that, use any technique to remove it from the pipeline.

- Also, it's normal to have empty stages in the middle not operating on any instruction after removing the instructions.
- In Packages 1 and 2, the execute stage takes 2 clock cycles.
  It is better (and I personally suggest it) to update the PC with the branch address (if the branch is taken) or the jump address after the second execute cycle and not after the first one.
  Meaning, let the branch/jump instruction finish its 2 clock cycles first in the execute stage, then if you are going to branch or jump, remove/flush the instructions in the decode and fetch stage from the previous cycle, and fetch the new instruction pointed by the updated PC.

Part 5: Conditional Branches PC

- There is 1 approach for handling the PC value when dealing with conditional branches/jumps:
- It is more accurate if you used the PC of the conditional branch instruction (storing it with the data of the instruction for later use).

Example:

Packages 3 and 4:

```
Cycles           1 2 3 4 5 6 7 8
0: BEQZ R1 5    F D E --> We stored the PC of BEQZ, thus: PC + 1 + IMM
= 0 + 1 + 5 = 6 if (R1 == 0)
1: ADD R2 R3     F D --> Removed
2: ADD R4 R5     F --> Removed
3: SUB R6 R7
4: SUB R8 R9
5: ADD R10 R11
6: ADD R12 R13        F D E
7: SUB R14 R15         F D E
8: SUB R16 R17           F D E
Total instructions that entered the pipeline: 6
Total cycles: 3 + (5 * 1) = 8 clock cycles
```

Packages 1 and 2:

```
Cycles           1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
21
0: BNE/JEQ R1 R2 3  F D D E E M W --> We stored the PC of BNE/JEQ,
thus PC + 1 + IMM = 0 + 1 + 3 = 4 if condition is true
1: ADD R2 R3 R4          F D D --> Removed
```

Project

General Description

```
2: ADD R5 R6 R7              F --> Removed
3: SUB R8 R9 R10
4: SUB R11 R12 R13               F D D E  E  M  W
5: ADD R14 R15 R16               F D  D  E  E  M  W
6: ADD R17 R18 R19                    F  D  D  E  E  M  W
7: SUB R20 R21 R22                       F  D  D  E  E  M  W
8: SUB R23 R24 R25                          F  D  D  E  E  M  W
Total instructions that entered the pipeline = 8
Total cycles = 7 + (7 * 2) = 21 clock cycles
```

Overflow vs Carry

Overflow

Overflow Occurs when the sign of the result can't be represented (needs 1 more bit).

- *Example: The result is 32 bits (max size of our data), but we need 1 more bit to represent the sign (33 bits).*
- *Note 1: There is a difference between the carry bit (useful in unsigned operations), and the overflow bit (useful in signed operations).*
- *Note 2: An overflow can occur, even if the carry is 0. They are different!*

Overflow Examples:

- Adding 2 positive numbers and the result is negative.
- Adding 2 negative numbers and the result is positive.

Quick tip to determine whether we have an overflow or not:

```
XOR the last 2 carries (from bit 6 to bit 7, and from bit 7)
XOR == 1 --> Overflow
XOR == 0 --> No Overflow
```

XOR Table:

```
0 XOR 0 = 0
0 XOR 1 = 1
1 XOR 0 = 1
1 XOR 1 = 0
```

Assume our values are represented in 8 bits only:

Example 1: Carry with Overflow

```
-128 in 8 bits = 1000 0000

-128 + -128 = -256 (result is negative)
      7654 3210   --> Bit Numbers
```

```
   1 0000 000     --> Carrys
     1000 0000    --> -128
    +                      --> +
     1000 0000    --> -128
    =                      --> =
     0000 0000  --> 0 INCORRECT (sign bit not represented in 8 bits,
need 1 more bit)
```

- In the above example, we have a carry of 1, and the result is 00000000 = 0.
- So, the result sign is positive while we were expecting a negative value.
- Thus, we have an overflow.
- Also, using our XOR method, the last 2 carries are 0 (from bit 6 to bit 7) and 1 (from bit 7)
- Therefore, 1 X0R 0 = 1 --> Overflow!

Example 2: Carry with No Overflow

```
64 in 8 bits = 0100 0000
-64 in 8 bits = 1100 0000

64 + -64 = 0

     7654 3210  --> Bit Numbers
  1  1000 000   --> Carrys
     0100 0000 --> 64
  +                     --> +
     1100 0000  --> -64
  =                     --> =
     0000 0000 --> 0 CORRECT (although we have a carry bit, we don't
have an overflow)
```

- The result is 0 which contains the expected sign.
- Using the XOR trick, the last 2 carries are 1 (from bit 6 to 7) and 1 (from bit 7).
- 1 XOR 1 = 0 (no overflow, although we have a carry bit!)

Example 3: Overflow with No Carry

```
64 + 64 = 128

     7654 3210 --> Bit Numbers
  0 1000 000   --> Carrys
     0100 0000 --> 64
  +                     --> +
     0100 0000 --> 64
  =                     --> =
     1000 0000 --> -128 INCORRECT (although we don't have a carry
bit, but we have an overflow)
```

- We added 2 positive numbers and the result is negative.
- We don't have a carry, however using the XOR trick, we have an overflow
- Carry from bit 6 to 7 is 1, while Carry from bit 7 is 0.
- 0 XOR 1 = 1 --> Overflow!
- Also, using our eyes, we can see the result having a negative sign although we are adding 2 positive numbers.

Carry Flag Check

- Since there are some special cases where UNSIGNED(R1) OP UNSIGNED(R2) > Byte.MAX_VALUE does not handle.
- The best solution to handle all cases is instead of checking whether the result is > Byte.MAX_VALUE or not, we can check on bit 8 (9th bit) of the result whether it is 1 or 0.
- We will still get the unsigned value for both operands, and check on the 9th bit (bit 8) of the result.

```
   9th bit (bit 8) = 1 --> Carry = 1
   9th bit (bit 8) = 0 --> Carry = 0
```

Example:

```
R5 = -5 = 0b11111011 in 8 bits
R6 = 5 = 0b00000101 in 8 bits
int temp1 = R5 & 0x000000FF = 0b00000000000000000000000011111011
int temp2 = R6 & 0x000000FF = 0b00000000000000000000000000000101
if( ((temp1 OP temp2) & MASK) == MASK) {
      Carry = 1;
} else {
      Carry = 0;
}
```

Project

General Description

Deliverables

The following guidelines must be followed in all packages:

Program Flow

a) You must write your program in **assembly language** in a text file.
b) Your must read the instructions from the text file, and parse them according to their types/formats (opcode and other relevant fields).
c) You must store the parsed version of the instructions in the memory (instruction segment of main memory or instruction memory according to your package).
d) You should start the execution of your pipelined implementation by fetching the first instruction from the memory (instruction segment of main memory or instruction memory) at Clock Cycle 1.
e) You should continue the execution based on the example provided in the Datapath section of each package reflecting the different stages working in parallel.
f) The Clock Cycles can be simulated as a variable that is incremented after finishing the required stages at a given time.
Examples:

```
Fetch();
Decode();
Execute;
// memory();
// writeback();

Cycle++
```

Printings

The following items must be printed in the console after each Clock Cycle:

a) The Clock Cycle number.
b) The Pipeline stages:
   - Which instruction is being executed at each stage?
   - What are the input parameters/values for each stage?
a) The updates occurring to the registers in case a register value was changed.
b) The updates occurring in the memory (data segment of main memory or data memory according to your package) in case a value was stored or updated in the memory.
c) The content of all registers after the last clock cycle.
d) The full content of the memory (main memory or instruction and data memories according to your package) after the last clock cycle.

Project

General Description

Package Selection

- You are requested to submit your selection of one of the previously described packages through the link:
- https://docs.google.com/forms/d/e/1FAIpQLSdQtnpaRiKgQepdfi5Z05ryYE-31za6AfOiOotof2AbWji76A/viewform?usp=dialog
- **Package Selection Deadline: Thursday 17 April 2025 at 11:59 pm**
  Note: Kindly note that the final assignment will be based on the first come first serve basis to ensure equal distribution of all packages among the teams
- You can check your assigned package through the link: https://docs.google.com/spreadsheets/d/1sQ7vqVayACci5kgdqKJyGAizTRFl8fLs7lf_5i8VGb4/edit?usp=sharing

Project Instructions

Please read the following instructions carefully:

a) Any case of plagiarism will result in a zero.
b) Any case of cheating will result in a zero.
c) A cheating detection tool will be used to compare the submitted projects against all online and offline implementations similar to the project idea.
   - The projects that have more than 50% similarity percentage will receive a zero.
d) It is your responsibility to ensure that you have:
   - Submitted before the deadline.
   - Submitted the correct file(s).
   - Submitted the correct file(s) names.

Submission Guidelines

- The submission deadline for submission is **Monday 19 May 2025 at 11:59 PM**
- You are requested to submit the following documents:
  1. A 1-min video to demonstrate the working code (please narrate and comment on the results)
     → **name the Video (Project_Team_m_Video.mp4)**
  2. The required project description report (kindly include in the cover page the team number, team name, package number and name, and team members' names, IDs, and tutorials).
     → **name the report (Project_Team_m_Report.pdf)**

3. The developed C code of the experiment, and any additional libraries/files that shall guarantee the execution of the file on the course team laptops during the evaluation with no problems in a single zip folder
   → **name the Code (Project _Team_m_Code.zip)**

- Please upload your milestone deliverables to your drive as a .zip file with the following naming format:
  (Ex.: CSEN601_S25_Proj_Package_**n**_Team_**m**.zip)
  *where **m** is your team number and **n** is your package number.*

- Submit **ONLY** the sharing link through the below form and ***Make sure that you give permission to access***
  https://docs.google.com/forms/d/e/1FAIpQLSdH43bKxA5E_OEhNTWF9X8HUNCfOJbsZBFV85OIMJCvmOoz0Q/viewform?usp=header

Evaluation Process

- The evaluation process of the process will be conducted during the first 2-3 days of the revision week.
- The evaluation timetable will be posted on the CMS during the last teaching week.

Project Grading

The project will be graded upon multiple criteria for each of the submitted deliverables. These criteria including *(below is just some grading items and more are considered):*

- The overall functionality of the project.
- Each technical aspect of the project will be graded as well.
- The quality of the submission (for example: well-commented and generic code, comprehensive and well-written reports, clear and comprehensive videos, and others).
- Submission on time with no delays (late submissions will be subject to deduction).
- The evaluation attendance is obligatory for all members and graded upon only the showing up.
- There is a collective team grade, yet during the evaluation and based on your discussion and answers, individual grades will be added as well.
- Note extra bonus marks will be added in case of successfully implementing Hazards handling (maximum bonus is 1.75% to be added to the total 100% course grade)