ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 222E
COMPUTER ORGANIZATION
PROJECT 2 REPORT

CRN : 21334

LECTURER : Doç. Dr. Gökhan İnce

GROUP MEMBERS:

150210005 : Arif Hilmi Yorulmaz

150210028 : Ertuğrul Şahin

SPRING 2024

Contents

1 INTRODUCTION

This report documents the design and implementation of a hardwired control unit for a custom CPU architecture. The control unit is based on the structural design from Part 4 of Project 1, and it aims to manage the execution of instructions stored in memory. The instructions are formatted in little-endian order and are loaded into the instruction register over two clock cycles due to the 8-bit output constraint of the RAM. The project involves handling two types of instructions: those with address references and those without. This includes various operations such as branching, arithmetic, logical shifts, and memory accesses. Our design leverages a combination of multiplexers, function selectors, and control signals to efficiently execute these instructions. The project culminates in a Verilog implementation, accompanied by simulation and memory files, ensuring that the control unit can fetch, decode, and execute instructions correctly from the starting memory address.

1.1 Task Distribution

Throughout the project, we always came together to advance the project, both in the research and learning process, in the coding and testing process, and the report writing process.

2 MATERIALS AND METHODS

2.1 Timing Unit Design

Our timing unit design ensures synchronized operation across the various components of the CPU by using a state counter (T) that advances with each clock cycle. Upon a reset signal (`SCReset`), the timing unit initializes the control signals to a default state. These include setting the register selectors for both the Register File (RF) and Address Register File (ARF) to inactive states, disabling the ALU write function, and disabling memory access (`Mem_CS`). The state counter (T) is then initialized to its starting state.

Once initialized, the timing unit operates by shifting the state counter left on each clock cycle, effectively advancing the state. When the counter reaches its final state (indicated by `T[7]`), it resets back to the initial state, ensuring a continuous loop through the predefined states. This mechanism provides a consistent and predictable timing framework that controls the sequence of operations within the CPU, allowing for precise coordination and synchronization of instruction execution.

2.2 Fetching: T[0] and T[1] Clock Cycles

In our project, we implemented the instruction loading process in two clock cycles, T[0] and T[1], due to the 8-bit output limitation of the RAM. This method ensures that both the least significant byte (LSB) and the most significant byte (MSB) of the instruction are correctly loaded into the instruction register (IR).

2.2.1 T[0] Clock Cycle

During the T[0] clock cycle, the LSB of the instruction is loaded from memory address A into the lower half of the IR. The necessary control signals are set to enable memory read mode, select the appropriate address from the Address Register File (ARF), and write the data to `IR[7:0]`.

2.2.2 T[1] Clock Cycle

In the T[1] clock cycle, the MSB of the instruction is loaded from memory address A+1 into the upper half of the IR. The control signals are configured to continue reading from memory and to write the data to `IR[15:8]`.

By the end of T[1], the complete 16-bit instruction is loaded into the IR, ready for execution in the subsequent cycles.

2.3 Instructions with Address Reference

Instructions with address reference follow the format shown in Figure 1. These instructions are structured with three primary fields:

- **The OPCODE**: A 6-bit field that defines the operation to be performed by the CPU.
- **The RSEL**: A 2-bit field used for register selection. This field determines which register from the Register File (RF) is used for the operation. The possible values for RSEL are `00`, `01`, `10`, and `11`, corresponding to registers $R1, R2, R3$, and $R4$ respectively.
- **The ADDRESS**: An 8-bit field that specifies the memory address involved in the operation. This address field is crucial for instructions that need to read from or write to specific memory locations.

The format for these instructions is represented as:

OPCODE (6-bit) RSEL (2-bit) ADDRESS (8-bit)

This structured format ensures that the CPU can efficiently decode and execute instructions that require both register and memory address references.

2.4 Instructions without Address Reference

Instructions without an address reference follow a different format. These instructions include several fields to specify the operation and the registers involved:

- **The OPCODE**: Like instructions with address references, these instructions also start with a 6-bit OPCODE field. This field specifies the operation to be performed.
- **The S**: A 1-bit field that indicates whether the condition flags will change as a result of the operation.
- **The DSTREG**: A 3-bit field that specifies the destination register. This field uses the values from 000 to 111 to select from registers like the Program Counter (PC), Address Register (AR), Stack Pointer (SP), and general-purpose registers $R1, R2, R3$, and $R4$.
- **The SREG1**: A 3-bit field that specifies the first source register. This field follows the same selection rules as DSTREG.
- **The SREG2**: A 3-bit field that specifies the second source register, following register selection rules as DSTREG and SREG1 do.

The format for these instructions is represented as:

OPCODE (6-bit) S (1-bit) DSTREG (3-bit) SREG1 (3-bit) SREG2 (3-bit)

This format allows the CPU to execute operations involving multiple registers without referencing a memory address, providing flexibility for arithmetic and logical operations within the CPU's registers.

2.4.1 Register Selectors

The CPU design includes two main sets of registers: the Register File (RF) and the Address Register File (ARF). The RF contains general-purpose registers $R1, R2, R3, R4$, and the ARF consists of the Program Counter (PC), Address Register (AR), and Stack Pointer (SP). The selection of which set to use depends on specific bits in the Instruction Register (IR).

The bits $IR[8]$, $IR[5]$, and $IR[2]$ are critical in determining which register set is used:

- If any bit is 1, the RF is used for its REG type (DSTREG, SREG1, SREG2).

- If any bit is 0, the ARF is used for its REG type (DSTREG, SREG1, SREG2)

This distinction is crucial for the CPU to know which component to fetch data from and write data to.

The destination register (DSTREG) is selected based on $IR[8]$:

- When $IR[8] = 1$, DSTREG is chosen from the RF, allowing for $R1, R2, R3$, or $R4$.
- When $IR[8] = 0$, DSTREG is chosen from the ARF, which includes the PC, AR, or SP.

The first source register (SREG1) is determined by $IR[5]$:

- When $IR[5] = 1$, SREG1 is selected from the RF ($R1, R2, R3$, or $R4$).
- When $IR[5] = 0$, SREG1 is selected from the ARF (PC, AR, or SP).

The second source register (SREG2) is chosen based on $IR[2]$:

- When $IR[2] = 1$, SREG2 is selected from the RF ($R1, R2, R3$, or $R4$).
- When $IR[2] = 0$, SREG2 is selected from the ARF (PC, AR, or SP).

After finding out from which source (RF or ARF) the registers will be selected, our progress strategy continues as follows: We look at the remaining bits for each 3-bit Register selector. That is, we select the necessary registers from the determined sources (RF or ARF) by looking at the IR[7:6] bits for DSTREG, IR[4:3] for SREG1, and IR[1:0] bits for SREG2.

The control signals, such as OutASel, OutBSel, FunSel, RegSel, and ScrSel for the RF, and OutCSel, OutDSel, FunSel, and RegSel for the ARF, manage the specific outputs, functions, and register selections within each register file. These signals ensure precise control over data flow and operations within the CPU.

In summary, the conditions of $IR[8]$, $IR[5]$, and $IR[2]$ determine whether the RF or ARF is used for fetching and writing data. This selection mechanism is essential for optimizing CPU performance and ensuring correct operation execution.

2.5 Operations

We preferred to explain operations by grouping them for a specific purpose. For example, if only SREG1 comes to the ALU, if SREG1 and SREG2 come to the ALU, and so forth. The main reason we did this was to increase the readability of the code we wrote when designing the CPU.

2.5.1 BRA: PC ← PC + VALUE

The provided code segment outlines the control logic for the Branch Always BRA operation, uniquely identified by the opcode 0x00, within a CPU's instruction cycle. This operation directs the Program Counter (PC) to jump to a new location determined by adding a specific value to the current PC. The sequence begins by setting up the Address Register File (ARF) and selecting appropriate inputs for the multiplexers that are instrumental in address computation or instruction decoding. Concurrently, the Register File (RF) is configured to retrieve the jump address or offset according to predefined function and source settings. As the sequence advances, adjustments are made to different multiplexer inputs to select the PC or associated values, indicating preparation for the jump computation. In the terminal computation phase, the Arithmetic Logic Unit (ALU) is programmed to perform the addition of the PC with the jump value, with the RF managing the data output directed into the ALU. Additional configurations in the ARF and RF refine the data routing and operational specifics. The temp register is set, likely serving as a temporary repository or control flag during this computation. This orchestrated setup across various hardware components ensures the precise update of the PC, facilitating the intended jump in program execution.

2.5.2 BNE & BEQ

Table 1: Conditional Branch Operations

Opcode	**Symbol**	**Operation**
0x01	BNE	IF $Z = 0$ THEN $PC \leftarrow PC + \text{VALUE}$
0x02	BEQ	IF $Z = 1$ THEN $PC \leftarrow PC + \text{VALUE}$

The operations 0x01 BNE and 0x02 BEQ are closely related conditional branch instructions in CPU architecture, both designed to modify the Program Counter (PC) based on the state of the zero flag (Z). While both instructions adjust the PC by adding a specified value (VALUE), they differ significantly in their activation conditions. The BNE (Branch if Not Equal) instruction executes the jump if Z is 0, meaning the previous operation did not result in zero. Conversely, the BEQ (Branch if Equal) performs the jump when Z is 1, indicating that the previous operation resulted in zero. This distinction allows the CPU to make conditional jumps based on different outcomes of computations, enhancing the control flow within programs by allowing operations to be dynamically skipped or executed based on runtime conditions.

2.5.3 POP

We select the SP register from ARF and increment it by 1. (T[2])

We read the value at this address in Read mode by sending the value in the SP register from ARF to memory as an address. We load the value we get from MemOut to Rx via MUXA (Only Write Low). In ARF, we select the SP register and increment it by 1. (T[3])

After loading the first 8 bits of Rx, we also load the last 8 bits (Only Write High) (T[4])

2.5.4 PSH

We select the SP register in ARF and apply Decrement with FunSel. We take the value in Rx directly from the ALU and send it to memory by taking the first 8 bits (Low) via MUXC. At the same time, we select the SP register from ARF via OutD and send it to memory, and by enabling Write mode, we write the value we received from MUXC to the SP address in memory. (T[2])

Finally, we perform the reprinting process by setting MuxCSel to receive the last 8 bits (High). (T[3])

2.5.5 INC & DEC

Table 2: Conditional Branch Operations

Opcode	**Symbol**	**Operation**
0x05	INC	$DSTREG \leftarrow SREG1 + 1$
0x06	DEC	$DSTREG \leftarrow SREG1 - 1$

There are two register selectors in these operations: SREG and DSTREG. When we look at the most significant bits of these selectors (IR[8] and IR[5]), it is obvious that we have 4 different scenarios in total since there are 2 components for each: RF and ARF. These 4 situations are:

- Source and destination are RF
- Source and destination are ARF
- Source is RF, destination is ARF
- Source is ARF, destination is RF

After the source and destination are determined, we adjust the FunSel of the system where the source is located according to the status of the operation. (000: Decrement, 001: Increment)

2.5.6 Shift Operations and NOT operation

The operations in this group are shift and NOT operations, which are performed with only one SREG (SREG1) coming to the ALU.

Table 3: Shift Operations and NOT operation

Opcode	**Symbol**	**Operation**
0x07	LSL	DSTREG ← LSL SREG1
0x08	LSR	DSTREG ← LSR SREG1
0x09	ASR	DSTREG ← ASR SREG1
0x0A	CSL	DSTREG ← CSL SREG1
0x0B	CSR	DSTREG ← CSR SREG1
0x0E	NOT	DSTREG ← NOT SREG1

Since there are 2 register selectors in each operation in this group, there are 4 scenarios in total.

If the source register is in the RF, we send the value in the register we selected from the RF via SREG to the A input of the ALU. We adjust the ALU FunSel according to the purpose of the operation we are processing. If the destination register is in RF, we send the value we receive from ALUOut back to RF via MUXA and load it into the destination register here. If the destination register is in ARF, we complete the similar process using MUXB and ARF. We complete all these processes in a single cycle.

If the source register is in ARF, we are unable to complete this in a single cycle. In the first cycle, we load the value in the register we selected with SREG from ARF, through MUXA via OutC, to S register (scratch register) in RF. We send the value in the S1 register to the A input of the ALU and adjust the ALU FunSel according to our operation. If the destination register is in RF, we perform the transition from MUXA and loading to RF, if the destination register is in ARF, we perform the transition from MUXB and loading to ARF (second cycle).

2.5.7 Logical and Arithmetic Operations

Table 4: Logical and Arithmetic Operations

Opcode	Symbol	Operation
0x0C	AND	DSTREG ← SREG1 AND SREG2
0x0D	ORR	DSTREG ← SREG1 OR SREG2
0x0F	XOR	DSTREG ← SREG1 XOR SREG2
0x10	NAND	DSTREG ← SREG1 NAND SREG2
0x15	ADD	DSTREG ← SREG1 + SREG2
0x16	ADC	DSTREG ← SREG1 + SREG2 + CARRY
0x17	SUB	DSTREG ← SREG1 - SREG2

Since there are 2 SREGs (SREG1 SREG2) and 1 DSTREG in this group, a total of 8 scenarios can occur.

One of the common features of the operations in this group is that the values in both source registers have already been loaded into the registers in the RF. If a source register is initially in the ARF, we load the value here to one of the Scratch registers in the RF. If it is already in the RF at the beginning, then we do not need to load this value into the RF additionally, we can get our value directly from the general purpose registers (R1, R2, R3, R4) in the RF in subsequent operations.

Another common feature is that the values in both source registers are processed through the A and B inputs of the ALU.

While writing code for the operations in this group during our CPU design process, we separated the SREGs according to whether they are connected to RF or ARF. Lastly, we examined the status of DSTREG. Thus, we handled the completion status of different scenarios in different cycle processes in an optimized manner.

We have considered the following 3 different situations so that the values in both source registers have already been loaded into the RF:

1. In case SREG1 and SREG2 target the registers in the RF, we complete our process in a single cycle (T[2]) in addition to T[0] and T[1], because both values we will send to the ALU are already in the RF.

2. In case one of SREG1 and SREG2 targets RF and the other targets ARF, in the first cycle (T[2]) we load our value in ARF to one of the Scratch registers in RF. In the second cycle (T[3]), we complete our process since both values are in RF.

3. In case SREG1 and SREG2 target the registers in the ARF, in the first cycle (T[2]), we load the value in the register in the ARF targeted by SREG1 into one of the

Scratch registers in the RF. We apply the same process for SREG2 in the second cycle (T[3]). In the third cycle (T[4]), we complete our process since both values are in RF.

In the last cycles of these 3 cases, we also adjust the ALU FunSel according to the type of operation by sending our values in RF to the A and B inputs of the ALU and load it into the register shown by DSTREG (RF or ARF).

The reason why we examine 3 different situations in this way is that each situation is completed in different cycles.

2.5.8 Operations with Flag Changes

Table 5: Operations with Flag Changes

Opcode	**Symbol**	**Operation**
0x18	MOVS	DSTREG ← SREG1, Flags will change
0x19	ADDS	DSTREG ← SREG1 + SREG2, Flags will change
0x1A	SUBS	DSTREG ← SREG1 - SREG2, Flags will change
0x1B	ANDS	DSTREG ← SREG1 AND SREG2, Flags will change
0x1C	ORRS	DSTREG ← SREG1 OR SRzEG2, Flags will change
0x1D	XORS	DSTREG ← SREG1 XOR SREG2, Flags will change

In fact, the processes taking place in the operations in this group are not systematically different from other operations. Here, different opcodes are given with the expression "flags will change" so that we can become familiar with such different architectures.

2.5.9 MOVL & MOVH

Table 6: MOVH and MOVL Operations

Opcode	**Symbol**	**Operation**
0x11	MOVH	DSTREG[15:8] ← IMMEDIATE (8-bit)
0x14	MOVL	DSTREG[7:0] ← IMMEDIATE (8-bit)

We take the first 8 bits of IR. For MOVL, we load the first 8 bits of Rx. For MOVH, we load the last 8 bits of Rx.

2.5.10 LDR: (16-bit) Rx ← M[AR] (AR is 16-bit register)

We select AR from OutD. We read it with read mode by sending the address to memory. Then we pass it through MUXA and load it into Rx.

2.5.11 STR: (16-bit) M[AR] ← Rx (AR is 16-bit register)

We select Rx from OutA and take it to the A input of the ALU. By setting FunSel to 10000, we obtain the value in Rx in ALUOut. We take AR from OutD and send it to memory as an address, enable write mode and write the first 8 bits of Rx to this address. We increase AR by 1. (T[2])

We take the AR that we previously incremented from OutD and send it back to the memory as an address and write the last 8 bits of Rx to this address. (T[3])

2.5.12 BL: PC ← M[SP]

The opcode 0x1F corresponds to the Branch and Link (BL) operation, which sets the Program Counter (PC) to the address stored in memory at the location pointed to by the Stack Pointer (SP).

We selected SP from ARF and sent it to memory. We put the memory's WR in reading mode by setting it to 0 and brought this value back to ARF with the MemOut output. We enabled the PC register in ARF and performed the Only Write Low operation with the 101 FunSel code. (T[2])

Then, we enabled the SP register in ARF and performed the increment operation with FunSel (T[3]).

Finally, we enabled the PC register again and wrote the remaining bits (Only Write High (T[4])

2.5.13 BX M[SP] ← PC, PC ← Rx

We load the value in the PC register we selected from ARF into the S1 register in RF. At the same time, with RF OutASel, we take the value in the desired register in the RF as the A input of the ALU, and by setting FunSel to 10000, we get the same value directly from the ALUOut. We load the value in ALUOut into the PC register of ARF. (T[2])

We send the SP register value we receive from ARF to the memory as an address via OutD and activate the Write mode by setting the WR' bit of the memory to 1. We take the value in the S1 register in the RF and take it directly to ALUOut with 10000 FunSel over the ALU. We pass the first 8 bits (Low) of the value we get from ALUOut through MUXC and write it to M[SP] and increase SP by 1. (T[3])

Finally, we write the last 8 bits (High) to M[SP] (SP that we have increased previously) via MUXC. (T[4])

2.5.14 LDRIM: Rx ← VALUE (VALUE defined in ADDRESS bits)

We pass the first 8 bits of IR through MUXA and put it in Rx.

2.5.15 STRIM: M[AR+OFFSET] ← Rx (AR is 16-bit register) (OFFSET defined in ADDRESS bits)

- We load AR into S1. (T[2])
- We load OFFSET to S2. (T[3])
- We collect S1 and S2 and load them into AR. (T[4])
- We assign the first 8 bits (Low) of Rx to M[AR] and increase AR by 1. (T[5])
- Add the remaining 8 bits of Rx (High) to M[AR]. (T[6])

3 Results

During the course of this project, we obtained various results that demonstrate the correct functionality and performance of our hardwired control unit for the custom CPU architecture. Below are the key results:

- **Instruction Execution**:
  - The control unit successfully executed a wide range of instructions, including branching, arithmetic, logical operations, and memory access instructions.
  - Each instruction was tested to ensure it produced the expected results and modified the appropriate registers and memory locations.
- **Timing and Synchronization**:
  - The timing unit ensured all operations were synchronized, with instructions executed over multiple clock cycles as required.
  - For example, the loading of the Instruction Register (IR) from memory was accurately performed over two clock cycles, as specified.
- **Functional Verification**:
  - We verified the correct functioning of all register selectors, ensuring the appropriate registers were selected based on the Instruction Register bits.
  - The ALU operations were tested to ensure they performed the correct arithmetic and logical operations, with the results correctly stored in the destination registers.

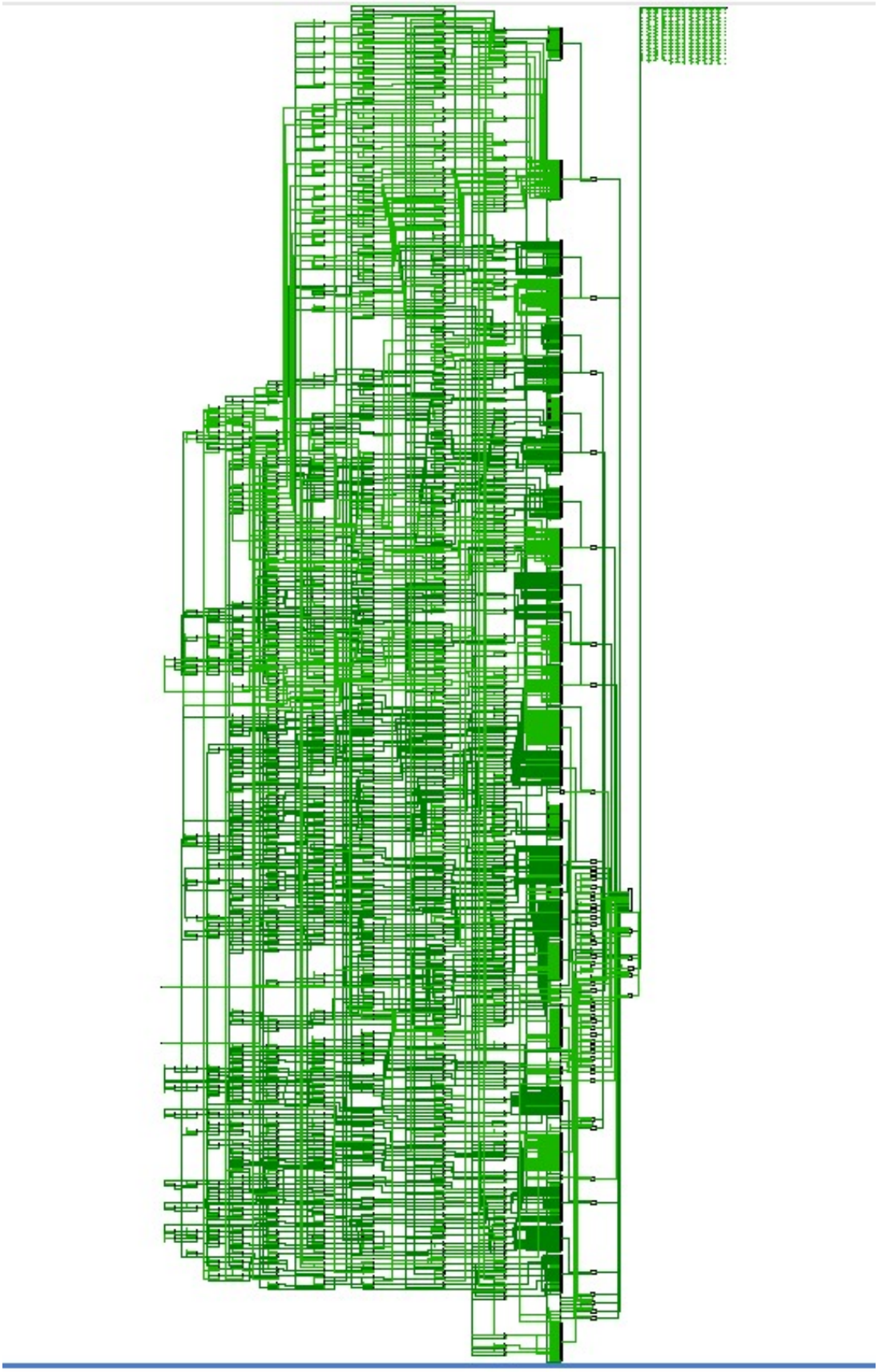

Figure 1: CPU

4 Operation Examples

4.1 BRA (Branch Always)

```
Output Values:
T:   1
Address Register File: PC:     0, AR:     0, SP:     0
Instruction Register :     x
Register File Registers: R1:     0, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:     x


Output Values:
T:   2
Address Register File: PC:     1, AR:     0, SP:     0
Instruction Register :     X
Register File Registers: R1:     0, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:     x


Output Values:
T:   4
Address Register File: PC:     2, AR:     0, SP:     0
Instruction Register :    40
Register File Registers: R1:     0, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:     x
```

Figure 2: BRA Operation Example - Image 1

```
Output Values:
T:   8
Address Register File: PC:     2, AR:     0, SP:     0
Instruction Register :    40
Register File Registers: R1:     0, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     2, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:     x


Output Values:
T:  16
Address Register File: PC:     2, AR:     0, SP:     0
Instruction Register :    40
Register File Registers: R1:     0, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     2, S2:    40, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:    42


Output Values:
T:   1
Address Register File: PC:    42, AR:     0, SP:     0
Instruction Register :    40
Register File Registers: R1:     0, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     2, S2:    40, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:    42
```

Figure 3: BRA Operation Example - Image 2

4.2 DEC (Decrement)

```
Output Values:
T:   1
Address Register File: PC:     0, AR:     0, SP:     0
Instruction Register :     x
Register File Registers: R1:    10, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:     x


Output Values:
T:   2
Address Register File: PC:     1, AR:     0, SP:     0
Instruction Register :     X
Register File Registers: R1:    10, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:     x


Output Values:
T:   4
Address Register File: PC:     2, AR:     0, SP:     0
Instruction Register :  6496
Register File Registers: R1:    10, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:    10
```

Figure 4: DEC Operation Example - Image 1

```
Output Values:
T:   8
Address Register File: PC:     2, AR:     0, SP:     0
Instruction Register :  6496
Register File Registers: R1:    10, R2:    10, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:    10


Output Values:
T:   1
Address Register File: PC:     2, AR:     0, SP:     0
Instruction Register :  6496
Register File Registers: R1:    10, R2:     9, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:    10
```

Figure 5: DEC Operation Example - Image 2

4.3 INC (Increment)

```
Output Values:
T:   1
Address Register File: PC:     0, AR:     0, SP:     0
Instruction Register :     x
Register File Registers: R1:    10, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:     x


Output Values:
T:   2
Address Register File: PC:     1, AR:     0, SP:     0
Instruction Register :     X
Register File Registers: R1:    10, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:     x


Output Values:
T:   4
Address Register File: PC:     2, AR:     0, SP:     0
Instruction Register :  5472
Register File Registers: R1:    10, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:    10
```

Figure 6: INC Operation Example - Image 1

```
Output Values:
T:   8
Address Register File: PC:     2, AR:     0, SP:     0
Instruction Register :  5472
Register File Registers: R1:    10, R2:    10, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:    10


Output Values:
T:   1
Address Register File: PC:     2, AR:     0, SP:     0
Instruction Register :  5472
Register File Registers: R1:    10, R2:    11, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:    10
```

Figure 7: INC Operation Example - Image 2

4.4 MOVH (Move High)

```
Output Values:
T:   1
Address Register File: PC:     0, AR:     0, SP:     0
Instruction Register :     x
Register File Registers: R1:    10, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:     x


Output Values:
T:   2
Address Register File: PC:     1, AR:     0, SP:     0
Instruction Register :     X
Register File Registers: R1:    10, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:     x


Output Values:
T:   4
Address Register File: PC:     2, AR:     0, SP:     0
Instruction Register : 17418
Register File Registers: R1:    10, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:     x
```

Figure 8: MOVH Operation Example - Image 1

```
Output Values:
T:   4
Address Register File: PC:     2, AR:     0, SP:     0
Instruction Register : 17418
Register File Registers: R1:    10, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:     x


Output Values:
T:   1
Address Register File: PC:     2, AR:     0, SP:     0
Instruction Register : 17418
Register File Registers: R1:  2570, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:     x
```

Figure 9: MOVH Operation Example - Image 2

4.5 MOVL (Move Low)

```
Output Values:
T:   1
Address Register File: PC:     0, AR:     0, SP:     0
Instruction Register :     x
Register File Registers: R1:    10, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:     x


Output Values:
T:   2
Address Register File: PC:     1, AR:     0, SP:     0
Instruction Register :     X
Register File Registers: R1:    10, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:     x
```

Figure 10: MOVL Operation Example - Image 1

```
Output Values:
T:   4
Address Register File: PC:     2, AR:     0, SP:     0
Instruction Register : 20490
Register File Registers: R1:    10, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:     x


Output Values:
T:   1
Address Register File: PC:     2, AR:     0, SP:     0
Instruction Register : 20490
Register File Registers: R1:    10, R2:     0, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:     x
```

Figure 11: MOVL Operation Example - Image 2

4.6 XOR (Exclusive OR)

```
T:   1
Address Register File: PC:     0, AR:     0, SP:     0
Instruction Register :     x
Register File Registers: R1:     9, R2:     6, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:     x


Output Values:
T:   2
Address Register File: PC:     1, AR:     0, SP:     0
Instruction Register :     X
Register File Registers: R1:     9, R2:     6, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:     x
```

Figure 12: XOR Operation Example - Image 1

```
Output Values:
T:   4
Address Register File: PC:     2, AR:     0, SP:     0
Instruction Register : 15781
Register File Registers: R1:     9, R2:     6, R3:     0, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:    15


Output Values:
T:   1
Address Register File: PC:     2, AR:     0, SP:     0
Instruction Register : 15781
Register File Registers: R1:     9, R2:     6, R3:    15, R4:     0
Register File Scratch Registers: S1:     0, S2:     0, S3:     0, S4:     0
ALU Flags: Z: x, N: x, C: x, O: x
ALU Result: ALUOut:    15
```

Figure 13: XOR Operation Example - Image 2

5 Discussion

Throughout this project, we implemented and refined our control unit design to manage the custom CPU architecture effectively. Here are the key points of analysis:

- **Design Considerations**:
  - We carefully considered the instruction format, ensuring that both address reference and non-address reference instructions were handled efficiently.
  - The use of specific bits in the Instruction Register to determine register selection allowed for flexible and efficient data management within the CPU.
- **Challenges and Solutions**:
  - One challenge was ensuring the correct synchronization of all components, especially when instructions required multiple clock cycles. This was addressed by designing a robust timing unit.
  - Another challenge was managing the various control signals needed to route data correctly through the ALU and registers. This was achieved through careful design and testing of the control logic.
- **Performance Analysis**:
  - The control unit's performance was evaluated based on its ability to execute instructions accurately and within the expected number of clock cycles.
  - The unit's efficiency in handling conditional branches (BNE, BEQ) and other complex operations (POP, PSH, INC, DEC) demonstrated its robustness and reliability.

6 Conclusion

This project provided valuable insights into the design and implementation of a hardwired control unit for a custom CPU architecture. We faced several challenges, particularly in ensuring synchronization and correct data routing, which were overcome through careful design and rigorous testing.

Key learnings from this project include:

- The importance of a well-defined instruction format and the role of the Instruction Register in managing CPU operations.

- The critical nature of timing and synchronization in ensuring the correct execution of instructions.
- The complexity involved in designing control logic to handle various CPU operations efficiently.

Overall, this project enhanced our understanding of CPU architecture and control unit design, providing a strong foundation for future work in computer organization and hardware design.

REFERENCES