Harvard VCPU With Arithmetic Shifts

Team 64

May 17, 2024

1 Team Members

- Peter Adel Makram , T-8, 55-4343
- Youssef Mostafa Faizallah, T-22, 55-2952
- Abdullah Hesham, T-22, 55-5012
- Youssef Raafat, T-22, 55-5066
- Ahmed Samy, T-23, 55-4895
- Seif Mohamed, T-6, 55-2821

2 Package

Package 4: Double Big Harvard combo large arithmetic shifts .

3 Purpose

Our goal is to implement a pipelined virtual central processing unit using a Harvard architecture. This VCPU supports arithmetic shifts, branching, logical and arithmetic instructions.

4 Introduction

4.1 Memory

In a harvard architecture, two memory units are used. One for instructions, and another one for data.

4.1.1 Instruction Memory

The instruction memory consists of 1024 words, each word is 16 bits long. For this purpose, the implementation uses a SHORT data type, which is defined in 16 bits.

4.1.2 Data Memory

The data memory consists of 2048 words, each word is 8 bits long. For this purpose, the implementation uses a CHAR data type, which is defined in 8 bits.

4.2 Register File

4.2.1 General Purpose Registers

Our CPU consists of 64 general purpose registers labeled $R_0 \rightarrow R_{63}$. Each register consists of 8 bits, so CHAR data type is used for the implementation as well.

4.2.2 Program Counter Register

Our program counter consists of 16 bits, so the SHORT data type is used for implementation.

4.2.3 Status Register

Our status register consists of 8 bits, so the CHAR data type is used for implementation. It consists of 5 flags, which are updated accordingly after specific instruction executions.

1. **Carry Flag** : Indicates when an arithmetic carry or borrow has been generated out of the most significant bit position.
2. **Two's Complement Overflow Flag** : Indicates when the result of a signed number operation is too large, causing the high-order bit to overflow into the sign bit.
3. **Negative Flag** : Indicates a negative result in an arithmetic or logic operation
4. **Sign Flag** : Indicates the expected sign of the result (not the actual sign).
5. **Zero Flag** : Indicates that the result of an arithmetic or logical operation was zero.

4.3 Instruction Set Architecture

The size of the instruction is 16 bits. There are two distinct instruction types, Register-Format & Immediate-Format. Both contain the opcode in the most significant 4 bits.

There are 12 distinct operations with opcodes ranging from 0 to 11. They include arithmetic, logical, memory access and branching operations.

We're faced with a serious limitation due to this since the immediate value is signed in logical and arithmetic and it consists of 6 bits. The range of values that can be represented using n bits is -2^{n-1} to $2^{n-1}-1$. So, with 6 bits, we can only use immediate values in the range -32 to 31. In memory access and branching operations, the immediate value is not signed, so the range is from 0 to 2^n-1, for 6 bits, 0 to 63. However, this limitation is improved within the registers as the 6 bit value is extended to 8 bits, so the range of values increase. In registers, the range of values supported is -128 to 127. Anything outside this range causes overflow, and automatically sets the overflow flag.

The status register is updated depending on the operation done.

1. **Carry Flag** : updated every ADD instruction.
2. **Overflow Flag** : updated every ADD or SUB instruction.
3. **Negative Flag** : updated every ADD, SUB, MUL, ANDI, EOR, SAL or SAR instruction.
4. **Sign Flag** : updated every ADD or SUB instruction.
5. **Zero Flag** : updated every ADD, SUB, MUL, ANDI, EOR, SAL or SAR instruction.

4.4 Datapath and Pipeline

4.4.1 Datapath

The datapath consists of 3 stages. **Fetch, Decode & Execute**. The execute does everything needed including memory access for loads and stores. All instructions execute in a single clock cycle.

4.4.2 Pipeline

For n instructions, the expected number of clock cycles is $3 + ((n-1) * 1)$. During the first clock cycle of any loading program, a single fetch is done to the first PC address. In the second clock cycle, the instruction fetched in the first clock cycle is decoded, and the next instruction is fetched. In the third clock cycle, the instruction fetched at the first clock cycle is executed, and the instruction fetched in the second clock cycle is decoded, and another instruction is fetched from the instruction memory.

5 Methodology

5.1 Initialization

During initialization of our program, a text file is read containing the instructions in assembly mnemonics. Each line read from the text file is parsed into its equivalent binary format, and then added to the instruction memory sequentially, line by line.

All the instructions execute on a common instruction memory, data memory, register file and pipeline blocks, so they're declared as global variables and initialized to contain zeros and nothing but zeroes.

5.2 Pipeline Initialization

A queue is used to represent the pipeline blocks. Two blocks are used to coordinate the pipeline behavior. The pipeline block separating the **fetch** and **decode** stages is a queue called ToBeDecodedQueue, which holds instructions that are fetched and need to be passed to the decode state later on. The pipeline block separating the **decode** and **execute** stages is called the ToBeExecutedQueue, which holds decoded instruction fields that need to be passed to the execute stage in the next clock cycle. A pipeline stages structure is used to manage which instruction is in which stage, and another pipeline structure is used to hold temporary fields and values of instructions that need to be passed to different stages so no loss of data happens.

We use two separate control hazard flags. First, controlHazardFlag which coordinates the pipeline execution after a BEQZ or a BR instruction in a way that ensures that no instructions in the pipeline execute afterwards, and that the instructions are being correctly fetched afterwards. A second flag, hasBranch is used to handle some base cases, such as having a branch to the last instruction of the file, or having a branch to the end of the file. It's automatically set to true after any branch operation.

5.3 Fetch Stage

In the fetch stage, an instruction is fetched from the instruction memory, then placed in ToBeDecodedQueue. The value of the PC is incremented.

The fetchInstruction() function returns the instruction (SHORT).

5.4 Decode Stage

In the decode stage, the instruction is obtained from the ToBeDecodedQueue, then decoded into all possible fields namely, the opcode, the source register, the destination register and the immediate value. The immediate value is also extended appropriately to 8 bits. Depending on the instruction, the immediate value is extended to be signed or unsigned accordingly. In memory access and branching instructions, the immediate value is extended so that it's unsigned. Otherwise, it's extended to be signed appropriately. This decoded instruction structure containing all decoded fields is then moved to the ToBeExecutedQueue, so that it can execute in the next clock cycle.

The decodeInstruction(short instruction) operates on the fetched instruction and returns a structure of decoded instruction fields that can be later on passed to the executeInstruction(decodedInstruction instruction) function.

5.5 Execute Stage

In the execute stage, the instruction fields are obtained from the ToBeExecutedQueue, then the opcode is inspected to determine the type of operation and the format of the instruction. The fields are then used appropriately to carry out the operation specified and make changes in the memory and register file accordingly.

The executeInstruction(decodedInstruction instruction) takes the structure of the instruction fields and operates on them. It returns void.

5.6 Program Flow

So, initially the file is read and parsed correctly. The instructions parsed are placed in the instruction memory sequentially. Then, the pipeline blocks are initialized along with the data memory, the instruction memory and the register file. Afterwards, the pipeline starts to execute the instructions in the behavior desired. Two functions are used in the main, namely loadProgram(char* filePath) which takes a file path to the instruction text file that it needs to read and parse. then, runProgram(), which continues executing instructions in the pipeline as desired.

After the program ends, the content of the register file are displayed along with the data memory.

5.7 Implementation

We only describe the methods, their return types, parameters and purpose in this section. For initialization, several methods are used for parsing the assembly into binary,

```
/* Used to initialize the memory units, the register file, and reading the
assembly text file. Every line that is read is passed onto a method that converts this line to
    binary */
void loadProgram(char* filePath);

/*This method divides the assembly instruction into its respective tokens, and uses several
    helper methods to obtain the binary value of the instruction and return it. */
short convertToBinary(char* line);

/* Helper methods used for retreiving the binary value */

char* getOpcodeBinary(char* opcode);
char* getRegisterBinary(char* registerName);
char* intToBinary(int num);
```

Datapath methods.

```
short fetchInstrunction();

decodedInstruction decodeInstruction(short currInstructionDecoded);

void executeInstruction(decodedInstruction decodedInst);
```

Pipeline methods.

```
void initializePipeline();

bool moveThroughPipeline();
```

6 Results

We're going to discuss some issues we faced during the implementation of this virtual CPU, a long with the output results of some assembly programs.

6.1 Immediate Value Parsing

For parsing the immediate value of 6 bits into binary when loading the program, initially, we transformed the number to binary normally. Which caused the immediate value to be always unsigned. We had to fix this issue by applying the rule of two's complement. When we got a negative immediate value, we negated the bits and added 1. Afterwards, the parsing was correct.

6.2 Bit Extension

In the decode stage, the immediate value is extended. We had to fix a similar issue of recognizing whether the MSB was 1 or not to determine whether it was signed or not, in order to extend it appropriately.

6.3 Decoded Instruction Fields

We had an issue with passing the values correctly to the execution stage in the pipeline. The field values were initially global variables, and they were overwritten by every stage. So, we had to create a structure that holds the decode fields and pass it on the execute stage appropriately to avoid this issue.

6.4 Control Hazard Flags

When we had a BR or BEQZ operation, the program correctly executed the instructions after the branch but we had an issue with the order of execution of the pipeline stages. So, we had to update some if-statement using the flag in the pipeline coordination method to ensure that instructions are being executed correctly afterwards.

6.5 Program Results

6.5.1 Arithmetic Instructions

```
Running Program,instructions not in the pipeline are labeled Instruction (stage): 0
-----------------------------------------------------
clock cycle: 1
Instruction  fetched: 1
Instruction  decoded: 0
Instruction  executed: 0
-----------------------------------------------------
clock cycle: 2
Instruction  fetched: 2
Instruction  decoded: 1
Instruction  executed: 0
-----------------------------------------------------
clock cycle: 3
Instruction  fetched: 3
Instruction  decoded: 2
Instruction  executed: 1
MOVI : R1 old Value : 0, Value in R1 after MOVI : 30
-----------------------------------------------------
clock cycle: 4
Instruction  fetched: 4
Instruction  decoded: 3
Instruction  executed: 2
MOVI : R2 old Value : 0, Value in R2 after MOVI : 30
-----------------------------------------------------
clock cycle: 5
Instruction  fetched: 5
Instruction  decoded: 4
Instruction  executed: 3
Status Register : 00000000
ADD : R1 Value : 30, R2 Value : 30, Value in Register 1 After Execution 60
-----------------------------------------------------
clock cycle: 6
Instruction  fetched: 6
Instruction  decoded: 5
Instruction  executed: 4
MOVI : R2 old Value : 30, Value in R2 after MOVI : 10
-----------------------------------------------------
clock cycle: 7
Instruction  fetched: 7
Instruction  decoded: 6
Instruction  executed: 5
Status Register : 00000000
SUB : R1 Value : 60, R2 Value : 10, Value in Register 1 After Execution 50
-----------------------------------------------------
clock cycle: 8
Instruction  fetched: 0
Instruction  decoded: 7
Instruction  executed: 6
Status Register : 00000000
SAL : R1 Value : 50, R1 Value after being shifted to the left 1 times : 100
-----------------------------------------------------
clock cycle: 9
Instruction  fetched: 0
Instruction  decoded: 0
Instruction  executed: 7
Status Register : 00000000
SAR : R1 Value : 100, R1 Value after being shifted to the right 1 times : 50
Program executed successfully ----------------------------------
```

This is the output for the assembly programming containing MOVI instructions and other arithmetic instructions.

- In clock cycle 5, the values in registers R1 and R2 are added correctly and the result is placed in R1.
- In clock cycle 7, the values in registers R1 and R2 are subtracted correctly and the result is placed in R1.
- In clock cycle 8, a SAR R1 1 multiplies the value of R1 by 2 correctly.
- In clock cycle 9, a SAL R1 1 divides the value of R1 by 2 correctly.

6.5.2 Logical Instructions

```
Running Program,instructions not in the pipeline are labeled Instruction (stage): 0
-----------------------------------------------------
clock cycle: 1
Instruction  fetched: 1
Instruction  decoded: 0
Instruction  executed: 0
-----------------------------------------------------
clock cycle: 2
Instruction  fetched: 2
Instruction  decoded: 1
Instruction  executed: 0
-----------------------------------------------------
clock cycle: 3
Instruction  fetched: 3
Instruction  decoded: 2
Instruction  executed: 1
MOVI : R1 old Value : 0, Value in R1 after MOVI : 30
-----------------------------------------------------
clock cycle: 4
Instruction  fetched: 4
Instruction  decoded: 3
Instruction  executed: 2
MOVI : R2 old Value : 0, Value in R2 after MOVI : 30
-----------------------------------------------------
clock cycle: 5
Instruction  fetched: 0
Instruction  decoded: 4
Instruction  executed: 3
Status Register : 00000001
EOR : R2 Value : 30, R1 Value : 30, Value in Register 2 After EOR 0
-----------------------------------------------------
clock cycle: 6
Instruction  fetched: 0
Instruction  decoded: 0
Instruction  executed: 4
Status Register : 00000000
ANDI : R1 Value : 30, Immediate Value : 30, Value in Register 1 After ANDI 30
Program executed successfully ----------------------------------
```

- In clock cycle 5, the value in register R2 and R1 are XOR'ed, since they're same value, the result is 0.
- In clock cycle 6, the value in register R1 and the immediate value are AND'ed, since they're the same value, the result is the same.

6.5.3 Memory Access Instructions

```
Running Program,instructions not in the pipeline are labeled Instruction (stage): 0
-----------------------------------------------------
clock cycle: 1
Instruction  fetched: 1
Instruction  decoded: 0
Instruction  executed: 0
-----------------------------------------------------
clock cycle: 2
Instruction  fetched: 2
Instruction  decoded: 1
Instruction  executed: 0
-----------------------------------------------------
clock cycle: 3
Instruction  fetched: 3
Instruction  decoded: 2
Instruction  executed: 1
MOVI : R1 old Value : 0, Value in R1 after MOVI : 25
-----------------------------------------------------
clock cycle: 4
Instruction  fetched: 0
Instruction  decoded: 3
Instruction  executed: 2
STR: Word in Register 1 : 25 , was loaded into memory at address 10
-----------------------------------------------------
clock cycle: 5
Instruction  fetched: 0
Instruction  decoded: 0
Instruction  executed: 3
LDA : Word in Memory Address 10 : 25, was loaded into Register 2
Program executed successfully ----------------------------------
```

- In clock cycle 4, the value in register R1 is stored in the data memory at address 10 correctly.
- In clock cycle 5, the value in memory at address 10 is loaded into register R2 correctly.

6.5.4 Branching Instructions

```
-----------------------------------------------------
clock cycle: 1
Instruction  fetched: 1
Instruction  decoded: 0
Instruction  executed: 0
-----------------------------------------------------
clock cycle: 2
Instruction  fetched: 2
Instruction  decoded: 1
Instruction  executed: 0
-----------------------------------------------------
clock cycle: 3
Instruction  fetched: 3
Instruction  decoded: 2
Instruction  executed: 1
MOVI : R1 old Value : 0, Value in R1 after MOVI : 0
-----------------------------------------------------
clock cycle: 4
Instruction  fetched: 4
Instruction  decoded: 3
Instruction  executed: 2
BEQZ : R1 Value : 0, Old PC Value : 4, Immediate Value : 2 ,New PC Value After BEQZ : 6
-----------------------------------------------------
clock cycle: 5
Instruction  fetched: 6
Instruction  decoded: 0
Instruction  executed: 0
-----------------------------------------------------
clock cycle: 6
Instruction  fetched: 7
Instruction  decoded: 6
Instruction  executed: 0
-----------------------------------------------------
clock cycle: 7
Instruction  fetched: 8
Instruction  decoded: 7
Instruction  executed: 6
MOVI : R1 old Value : 0, Value in R1 after MOVI : 5
-----------------------------------------------------
clock cycle: 8
Instruction  fetched: 0
Instruction  decoded: 8
Instruction  executed: 7
MOVI : R2 old Value : 0, Value in R2 after MOVI : 5
-----------------------------------------------------
clock cycle: 9
Instruction  fetched: 0
Instruction  decoded: 0
Instruction  executed: 8
Status Register : 00000000
MUL : R1 Value : 5, R2 Value : 5, Value in Register 1 After Execution 25
```

- In clock cycle 4, a BEQZ updates the PC value appropriately.
- In clock cycles 5 and 6, the previous instructions in binary remaining in the toBeDecodedQueue are flushed, and the decoded instructions in the toBeExecutedQueue are flushed. So we have to start the pipeline from the start, with the instructions at the new PC value.
- In clock cycles 7 through 9, the instructions after the BEQZ are executed correctly.