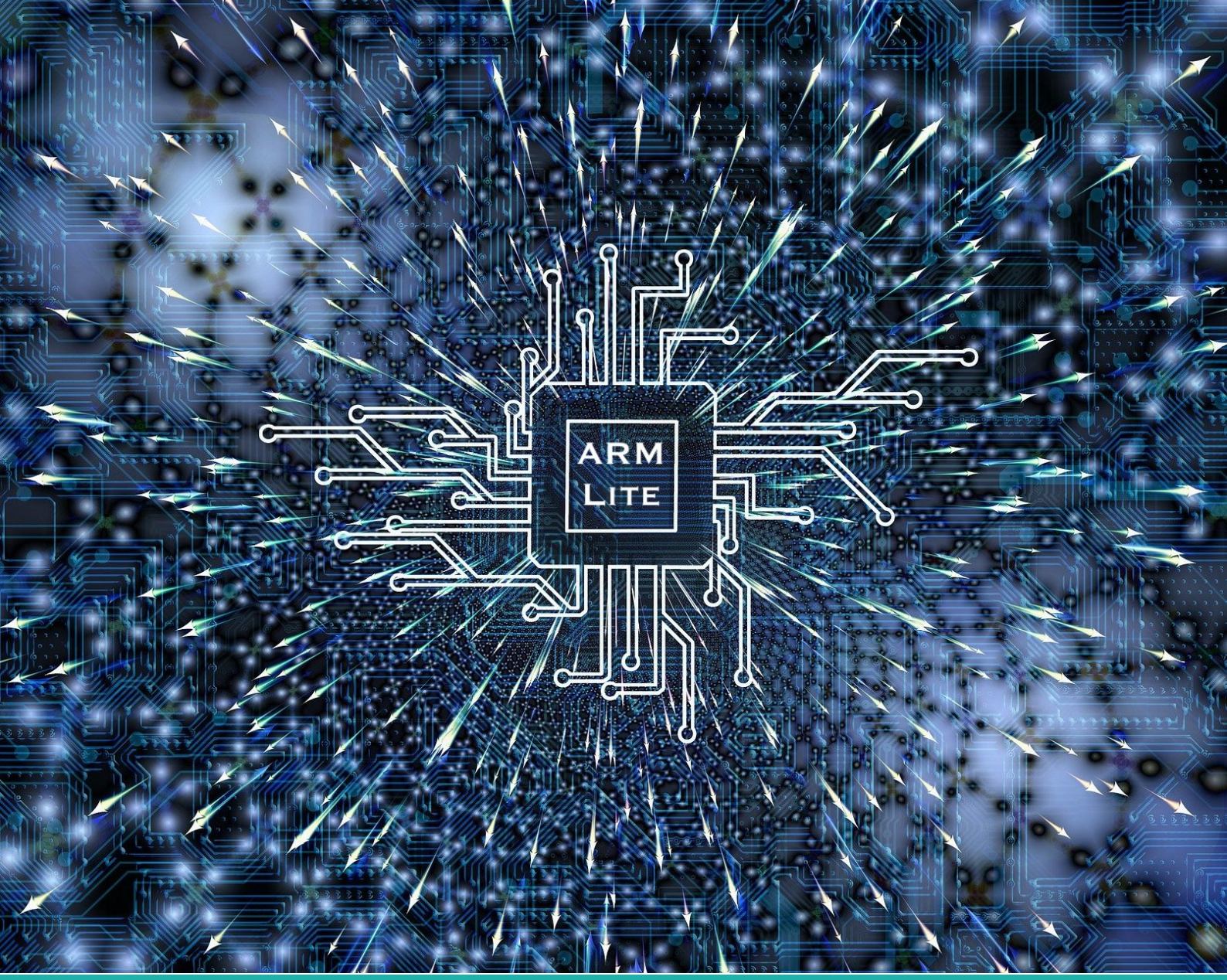



# ARM Lite: Pipelined CPU

## Verilog Implementation with Hazard Detection and Forwarding

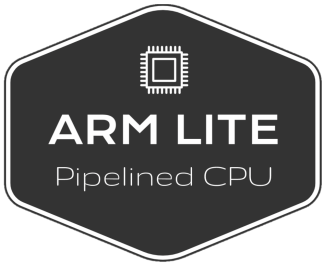
**Author:** Wuqiong Zhao & Ruiqi Zheng

**Institute:** Southeast University

**Date:** Jan. 7, 2022

**Version:** 2.0






*Project Website:* *arm-lite.teddy-van-jerry.org*

# Contents

# Preface

Before you comb through the contents of **ARM Lite** CPU design, a brief understanding of systematic design work is like can be beneficial. The impression of developing this project is that it is by no means easy but the whole system we build rewards us with great fulfillment. This project features the pipelined CPU with hazard detection and forwarding, which leads to a lot of difficulty in implementation.

The project uses `Verilog` on Xilinx Vivado 2017.4 and has been tested both on Ubuntu and Windows. Apart from the hardware implementation of the CPU, the project also contains a compiler that translates ARM (64-bit) code into machine code using `C++`.

The authors would like to thank Man Feng and Xudong Ma for their assistance for their guidance and assistance in pipelined CPU design. This work is originally the course project for Digital Design and Computer Architecture, 2021 Fall of Southeast University.

There may be errors in this project and you are welcome to pull request on GitHub or contact the author via E-Mail: wqzhao@seu.edu.cn.

Wuqiong Zhao
Jan. 7, 2022
At Southeast University

# Chapter 1 Brief Introduction

**Introduction**

- ❏ *GitHub Repository*
- ❏ *Official Website*
- ❏ *Project License*
- ❏ *Project Aim*

## 1.1 Project Host Information

The project of ARM Lite is open source and hosted on GitHub: github.com/Teddy-van-Jerry/ARM_Lite with the official website at arm-lite.teddy-van-jerry.org.

You can fork the project repository or download source code at the release page. Since this project is licensed under thr MIT License, you are free to use it subject to the requirements.

## 1.2 Project Aim

Pipelined CPU design has proven its vitality in many ways, as is elaborated in [1, 2]. Moreover, [3] points out that it is especially a good training project for teaching computer architecture. Among Reduced Instruction Set Computer (RISC) architectures, ARM has a good performance in parallelism shown in [4] which enables its thrive today together with X86_64. The instruction set of ARM can be regarded as a compromise. The supported instructions include the most frequently used ones and also less popular instructions. This makes ARM instructions not so long but still able to be pipelined effectively.

Therefore we choose the pipelined CPU with ARM instructions set and implements its most basic instructions with R, I, D, B and CB types following the ideas of [5].

# Chapter 2 System Design

**Introduction**

- ❏ *Pipeline Design*
- ❏ *Forwarding*
- ❏ *Hazard Detection*

## 2.1 Pipeline Design

The detailed design with control signals is shown in Fig. 2.1 where the bold black line represents the data flow while the thin gray line represents the control signals.

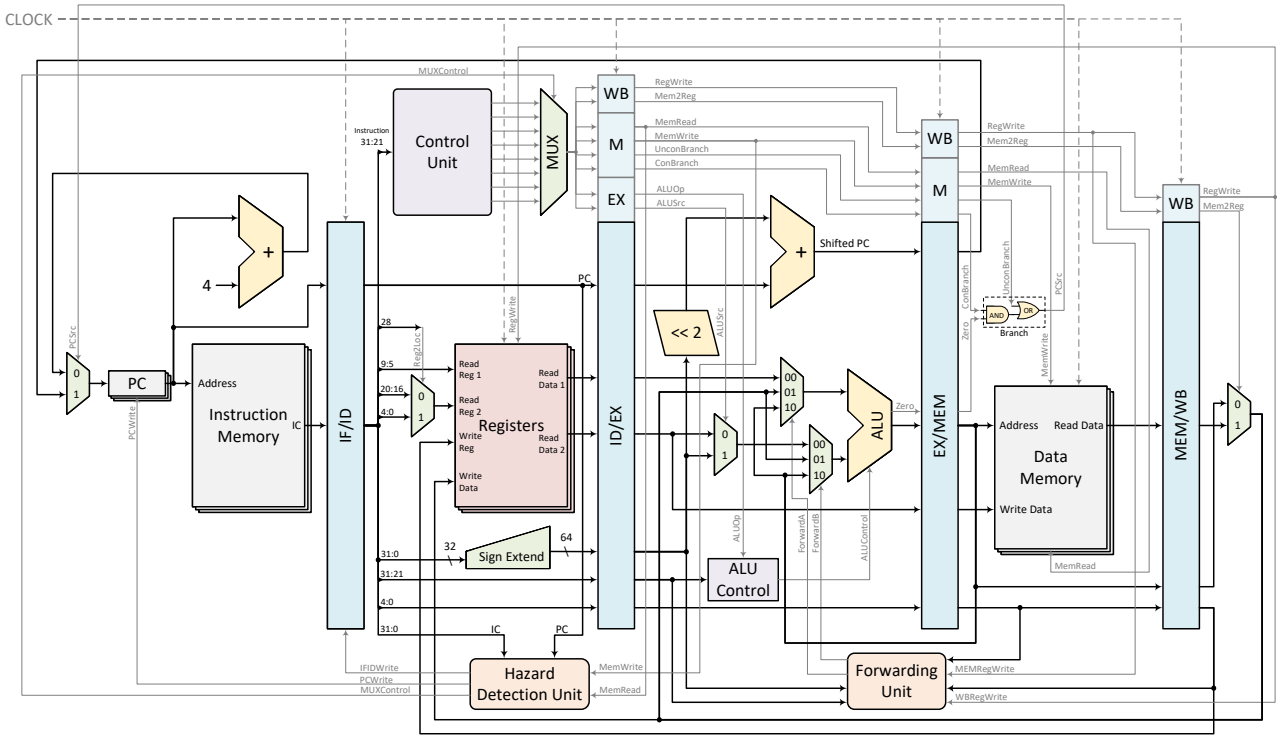



**Figure 2.1:** ARM Lite CPU design, detailing Fig. 4.63 of [5].

There are five periods in this CPU, Instruction Fetch (IF)Instruction Decode (ID), Execute (EX), Memory Access (MEM) and Register write back (WB) which are shown clearly in Fig. 2.1, divided by a large register between periods. The functions of each components will be elaborated in Chapter 3.

## 2.2 Hazard Detection and Forwarding

Hazard Detection and Forwarding are both modules that work together to solve the hazard problem.

# Chapter 3 Supported Instructions

**Introduction**

- ❑ *R Type Instructions*
- ❑ *I Type Instructions*
- ❑ *D Type Instructions*
- ❑ *B Type Instructions*
- ❑ *CB Type Instructions*
- ❑ *Data Flow Graphs*

## 3.1 Summary

Apart from the instructions already supported by the original project, this project adds all I type and CB type instructions and also extend the range of R type instructions like `LSL`, `LSR` and `EOR`.

## 3.2 R Type

### 3.2.1 ADD

Example: `ADD r5, r3, r2` (Add the values of `r3` and `r2` then put the result into `r5`)

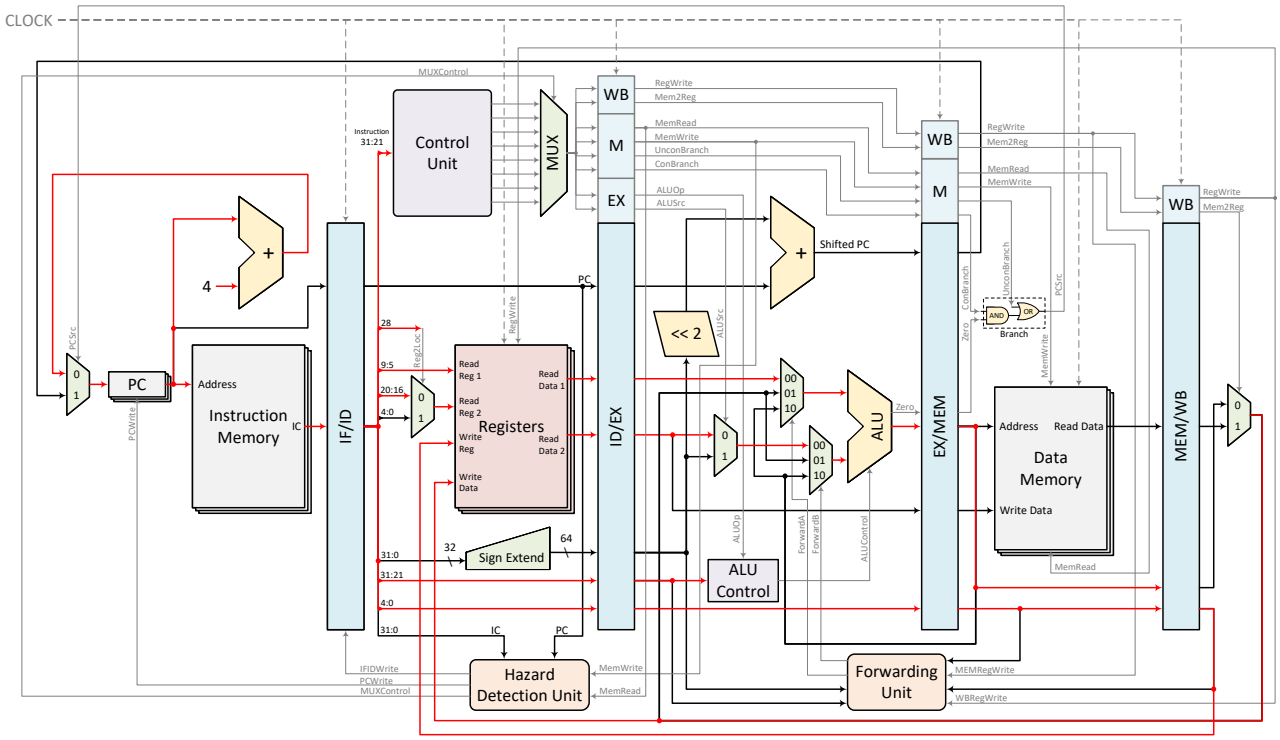



**Figure 3.1:** Data flows for instructions `ADD`, `SUB`, `AND`, `ORR`, `EOR`.

- `PC` will be incremented by 4 and the shifted `PC` will be discarded.
- Input registers are in `[20:16]` and `[9:5]`.
- `ALU MUX` chooses 0, i.e. data just read from the registers.
- `ALUControl` will tells `ALU` to do `ADD` operation.
- No need to write memory but need to write back to registers.

### 3.2.2 SUB

Example: `SUB r4, r3, r2` (Subtract the values of `r3` and `r2` then put the result into `r4`)

- This is similar to `ADD` instruction except for the operation given by `ALU Control` to `ALU`.

### 3.2.3 AND

Example: `AND r4, r3, r2` (Bit-wise AND the values of `r3` and `r2` then put the result into `r4`)

- This is similar to `ADD` instruction except for the operation given by `ALU Control` to `ALU`.

### 3.2.4 ORR

Example: `ORR r6, r2, r3` (Bit-wise OR the values of `r2` and `r3` then put the result into `r6`)

- This is similar to `ADD` instruction except for the operation given by `ALU Control` to `ALU`.

### 3.2.5 EOR

Example: `EOR r6, r2, r3` (Exclusive OR the values of `r2` and `r3` then put the result into `r6`)

- This is similar to `ADD` instruction except for the operation given by `ALU Control` to `ALU`.

### 3.2.6 LSL

Example: `LSL r6, r2, #3` (Logical Shift Left the values of `r2` by 3 then put the result into `r6`)

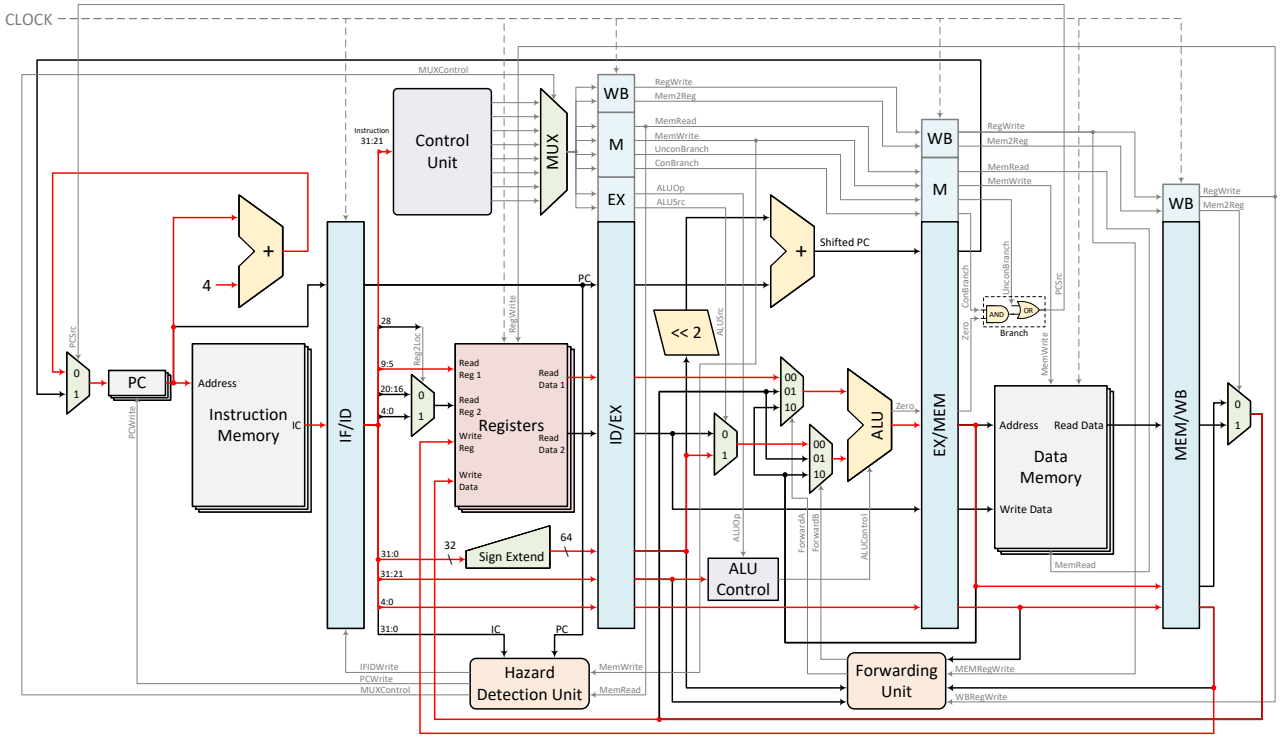



**Figure 3.2:** Data flows for instructions `ADDI`, `SUBI`, `ANDI`, `ORRI`, `EORI`, `LSL`, `LSR`.

- `PC` will be incremented by 4 and the shifted `PC` will be discarded.
- Input register is in `[9:5]`.
- `ALU MUX` chooses `1`, i.e. data coming from `Sign Extend`, the `shamt` part in the machine code.

- `ALUControl` will tells `ALU` to do `LSL` operation.
- No need to write memory but need to write back to registers.

### 3.2.7 LSR

Example: `LSR r6, r2, #3` (Logical Shift Right the values of `r2` by 3 then put the result into `r6`)

- This is similar to `LSL` instruction except for the operation given by `ALU Control` to `ALU`.

## 3.3 I Type

### 3.3.1 ADDI

Example: `ADDI r5, r3, #2` (Add the values of `r3` and immediate 2 then put the result into `r5`)

- Different from instruction `ADD`, the immediate comes from the `Sign Extend` and the `ALU MUX` chooses the immediate out value.

### 3.3.2 SUBI

Example: `SUBI r4, r3, #2` (Subtract the values of `r3` with immediate 2 then put the result into `r4`)

- This is similar to `ADDI` instruction except for the operation given by `ALU Control` to `ALU`.

### 3.3.3 ANDI

Example: `ANDI r4, r3, #2` (Bit-wise AND the values of `r3` and 2 then put the result into `r4`)

- This is similar to `ADDI` instruction except for the operation given by `ALU Control` to `ALU`.

### 3.3.4 ORRI

Example: `ORRI r6, r2, #3` (Bit-wise OR the values of `r2` and 3 then put the result into `r6`)

- This is similar to `ADDI` instruction except for the operation given by `ALU Control` to `ALU`.

### 3.3.5 EORI

Example: `EORI r6, r2, #3` (Exclusive OR the values of `r2` and 3 then put the result into `r6`)

- This is similar to `ADDI` instruction except for the operation given by `ALU Control` to `ALU`.

## 3.4 D Type

### 3.4.1 LDUR

Example 1: `LDUR r2, [r10]` (Retrieve the value in memory at location `r10` and put that value into register `r2`)

Example 2: `LDUR r3, [r10, #1]` (Retrieve the value in memory at the location `r10` + immediate (1) and put that value into register `r3`)

- `PC` will be incremented by 4 and the shifted `PC` will be discarded.
- Input register is in `[9:5]`.

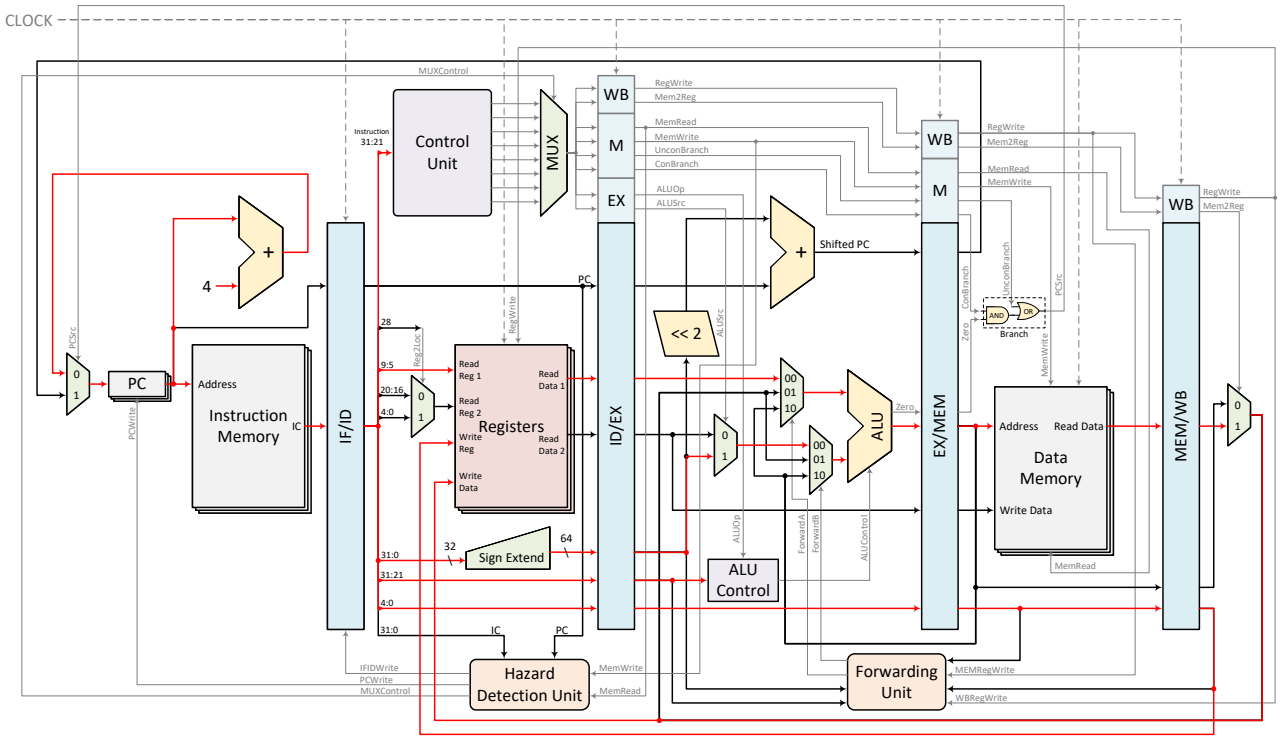



**Figure 3.3:** Data flow for instruction `LDUR`.

- `ALU MUX` chooses `1`, i.e. data coming from `Sign Extend`, the addition to register value. Then the result of them will be the data to write back.
- `ALUControl` will tells `ALU` to do `ADD` operation.
- No need to write memory but need to write back to registers.
- It should be noticed that not until the end of period `MEM` the result is not given out. This is different from instructions of R or I types where the calculation result comes immediately after `ALU`, i.e. the period of `EX`.

### 3.4.2 STUR

Example 1: `STUR r1, [r9]` (Store the value of `r1` into memory at the location `r9`)

Example 2: `STUR r4, [r7, #1]` (Store the value of `r4` into memory at the location `r7` + immediate (1))

- `PC` will be incremented by 4 and the shifted `PC` will be discarded.
- Input register is in `[9:5]`.
- `ALU MUX` chooses `1`, i.e. data coming from `Sign Extend`, the addition to register value. Then the result of them will be the data to write back.
- `ALUControl` will tells `ALU` to do `ADD` operation.

## 3.5 B Type

### 3.5.1 B

Example: `B #2` (Jump instruction with 2)

- With the Unconditional Jump signal, the `PC` is set to the shifted `PC`.

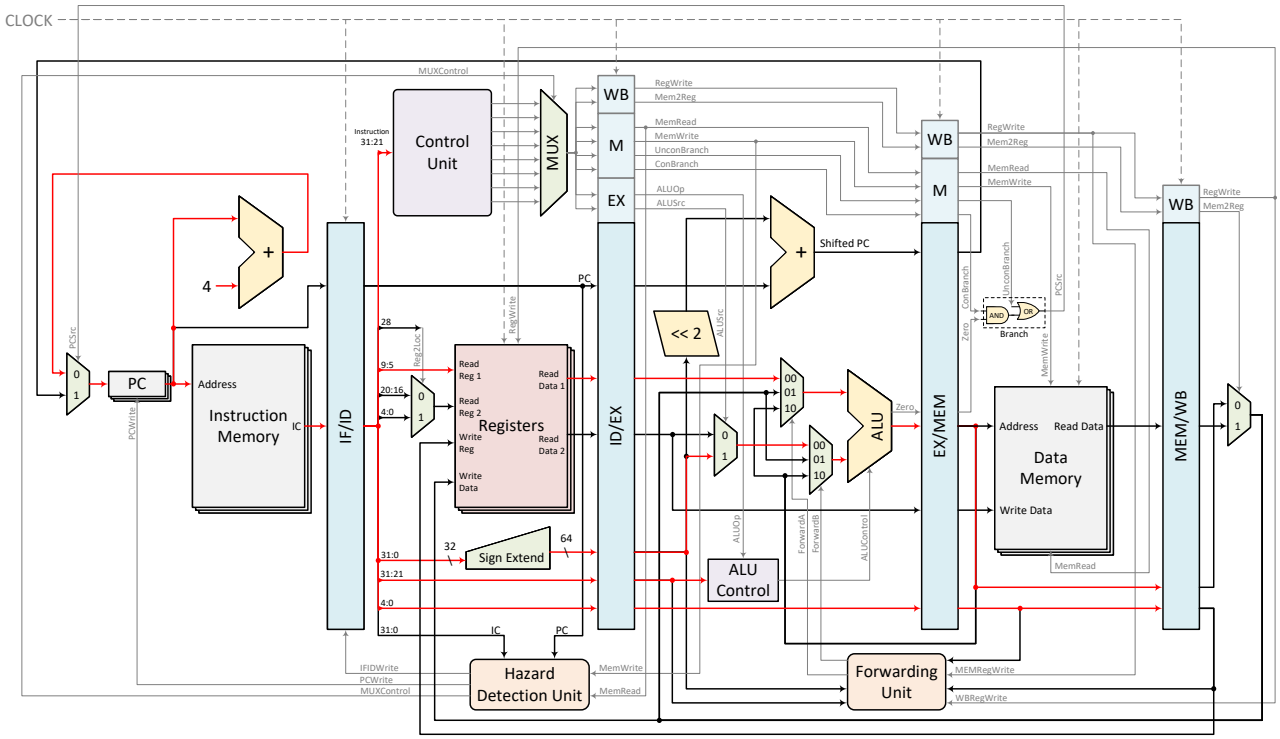



**Figure 3.4:** Data flow for instruction STUR.

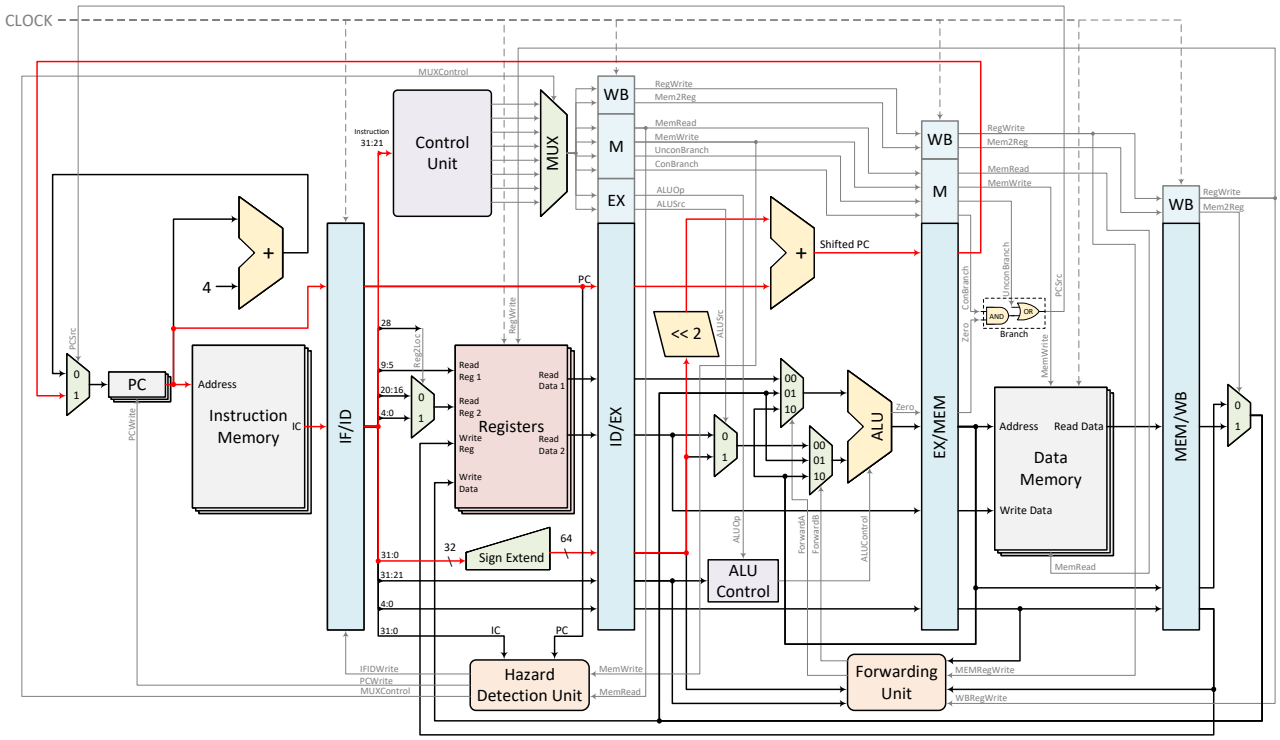



**Figure 3.5:** Data flow for instruction B.

## 3.6 CB Type

### 3.6.1 CBZ

Example: `CBZ r1, #2` (If the value of `r1` is zero then jump to instruction 2, otherwise, continue executing `PC++`)

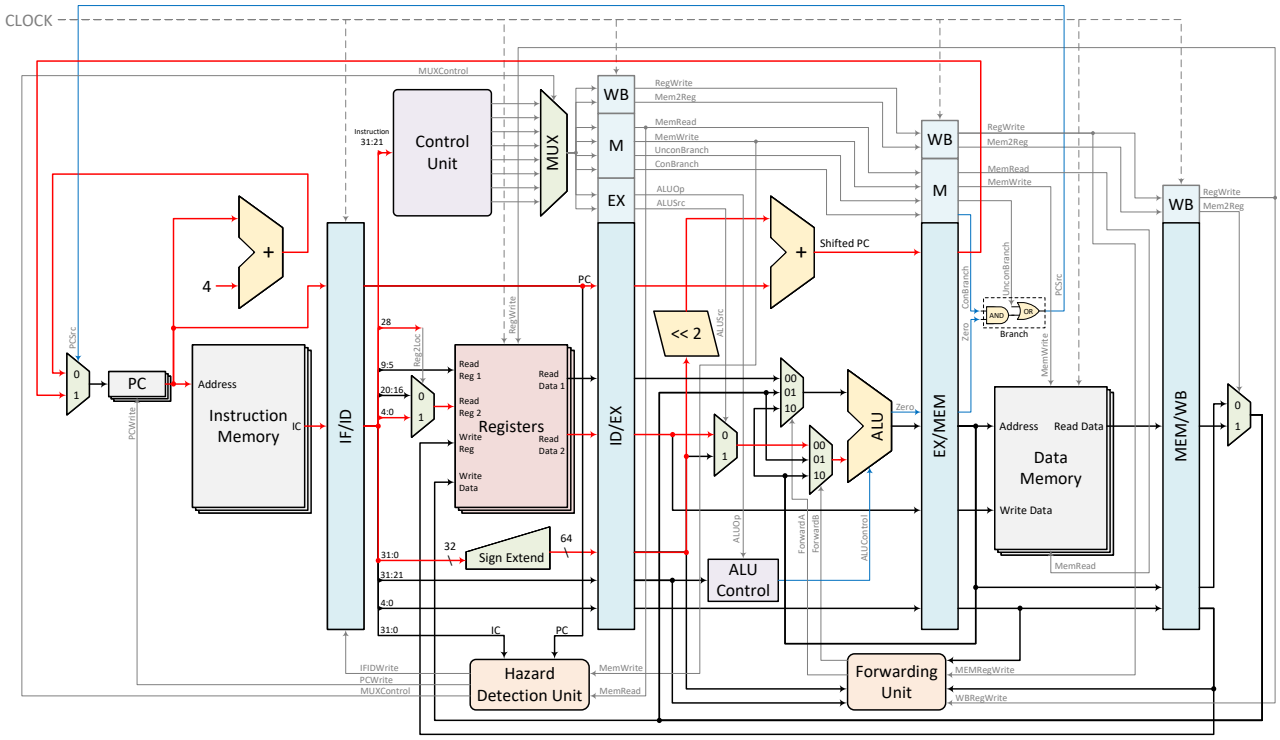



**Figure 3.6:** Data flows for instructions `CBZ`, `CBNZ`.

- The Conditional Branch is set to `1`.
- The most important thing here is the `Zero` flag of `ALU`. The operation of `ALU` is takes the value of `[4:0]` directly. Therefore, the result of the `Branch` module generates the `PCSrc` signal that decides whether `PC` to jump.

### 3.6.2 CBNZ

Example: `CBNZ r1, #2` (If the value of `r1` is not zero then jump to instruction 2, otherwise, continue executing `PC++`)

- The Conditional Branch is set to `1`.
- The most important thing here is the `Zero` flag of `ALU`. The operation of `ALU` is takes the logic Not of `[4:0]`. Therefore, the result of the `Branch` module generates the `PCSrc` signal that decides whether `PC` to jump.

# Chapter 4 Simulation Examples

**Introduction**

- *Behavioral Simulation*
- *Simulation Waveforms*
- *Instructions Analysis*

## 4.1 Example 1

```
LDUR  x0, [x2, #3]  ; f8403040
ADDI  x5, x9, 7     ; 91001d25
ANDI  x3, x3, 3     ; 92000c63
SUBI  x6, x6, 3     ; d1000cc6
LSL   x7, 1         ; d36004e7
B     -3            ; 17fffffd
```

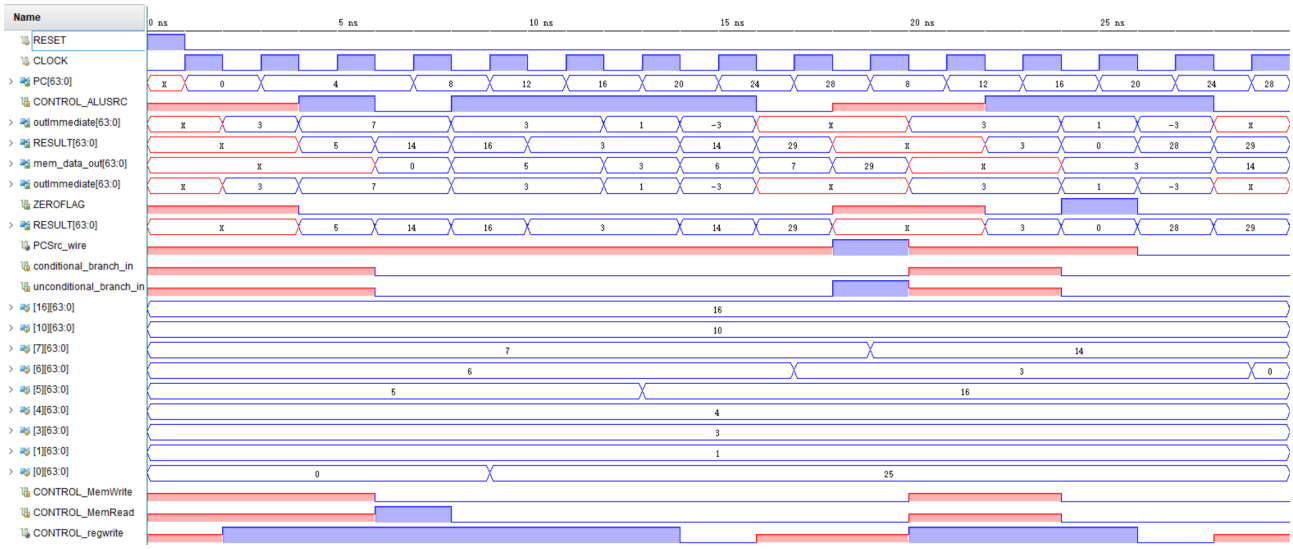



**Figure 4.1:** Simulation Waveforms for Example 1

With the IF, ID, EX, MEM and WB stages of a pipelined CPU, each instruction is executed at the positive edge of the clock signal. Therefore, there will be five occurrences of the positive edges during one cycle of an instruction.

In example 1, six instructions are used.

During the `LDUR` instruction, `PC=0`. When `PC=4`, `x2` is read from the registers. This value is then added with the immediate value from `Sign Extend`. (Because of the hazard detection, one `NOP` is added automatically.) From the third positive edge, `ALU` executes the addition of `x2` and `3`. In the fourth positive edge, the value of `RAM[5]` is read out (We can find the `MemRead` control signal during 6-8 ns). From the fifth positive edge, `x1` and `RAM[5]` is returned to the registers and then `x1=RAM[5]`. Because of `NOP`, `ADDI` instructions come from the the third positive edge and write back at the 8th positive edge. When dealing with immediate values, `ALUSrc` signal is set to 1.

It is similar for `ANDI`, `SUBI`, `LSL` commands, and the changes of values in registers can also be observed.

Because the jump by instruction `B` occurs in the fourth clock period, `PC` jumps from 28 to 8, forming a loop. When the jump is required, `PCSrc` is set to 1.

## 4.2 Example 2

```
LDUR  x0 , [x2, #3] ; f8403040
LSR   x16, #1       ; d3400610
EOR   x1 , x2 , x0  ; ca000041
CBZ   x31, #8       ; b400011f
ORRI  x4 , x5 , 7   ; b2001ca4
EORI  x10, x11, #11 ; d2002d6a
; SKIP THREE LINES ;
ORRI  x4 , x5 , 7   ; b2001ca4
EORI  x10, x11, #11 ; d2002d6a
STUR  x11, [x5, #6] ; f80060ab
```


**Figure 4.2:** Simulation Waveforms for Example 2

Here we elaborate on the conditional branch `CBZ` instruction. First the `ZERO` flag is 1 in 12-14 ns and then conditional branch signal is 1 in 14-16 ns. Therefore, the branch will get the result 1 in 16-18 ns. In the next clock period, the `PCSrc` will be set as 1 and then the conditional jump succeeds.

# Chapter 5 Known Issues

## 5.1 CPU Project

- Hazard Detection and Forwarding is naive and cannot tell exactly tell where forwarding or `NOP` is needed.
- Control hazard can not be solved so far and can lead to fatal error in simulation.

## 5.2 Compiler

This work is still under progress and should be finished in the following weeks.

# Chapter 6 Version History

We revised our project now and then. This section shows the version story of ARM Lite. We have 37 commits and 2 releases on GitHub.

---

**2022/01/07** *Updates:release of v2.0*

① Solve the bug with instructions `CBZ` and `CBNZ`.

② Add Simulation Test.

---

**2021/12/24** *Updates:release of v1.1*

① Add instructions.

---

**2021/12/23** *Updates:release of v1.0*

① Work based on ARM-LEGv8 (pipelined with hazard detection and forwarding) is implemented on Xilinx Vivado 2017.4 both on Ubuntu and Windows.

② Create the Visio Drawing for the CPU architecture.

# Bibliography


[1] G. S. Sohi, "Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers," vol. 39, no. 3, pp. 349–359, 1990.

[2] G. Qin, Y. Hu, L. Huang, and Y. Guo, "Design and performance analysis on static and dynamic pipelined cpu in course experiment of computer architecture," in *Proceedings of IEEE International Conference on Computer Science & Education (ICCSE)*, IEEE, 2018, pp. 1–6.

[3] J. H. Lee, S. E. Lee, H. C. Yu, *et al.*, "Pipelined CPU design with FPGA in teaching computer architecture," vol. 55, no. 3, pp. 341–348, 2011.

[4] J. Goodacre and A. N. Sloss, "Parallelism and the ARM instruction set architecture," *Computer*, vol. 38, no. 7, pp. 42–50, 2005.

[5] D. A. Patterson and J. L. Hennessy, *Computer organization and design ARM edition: the hardware software interface*. Morgan kaufmann, 2016.

[6] C. T. Carlstrom and T. S. Fuerst, "Agency Costs, Net Worth, and Business Fluctuations: A Computable General Equilibrium Analysis," *The American Economic Review*, pp. 893–910, 1997, ISSN: 0002-8282.

[7] Q. Li, L. Chen, and Y. Zeng, "The Mechanism and Effectiveness of Credit Scoring of P2P Lending Platform: Evidence from Renrendai.com," *China Finance Review International*, vol. 8, no. 3, pp. 256–274, 2018.

# Appendix A Future Work

Issues listed will be dealt with and a better documentation will be done together with the online documentation.

Moreover, an essay about the design of ARM Lite CPU will also be on the agenda.

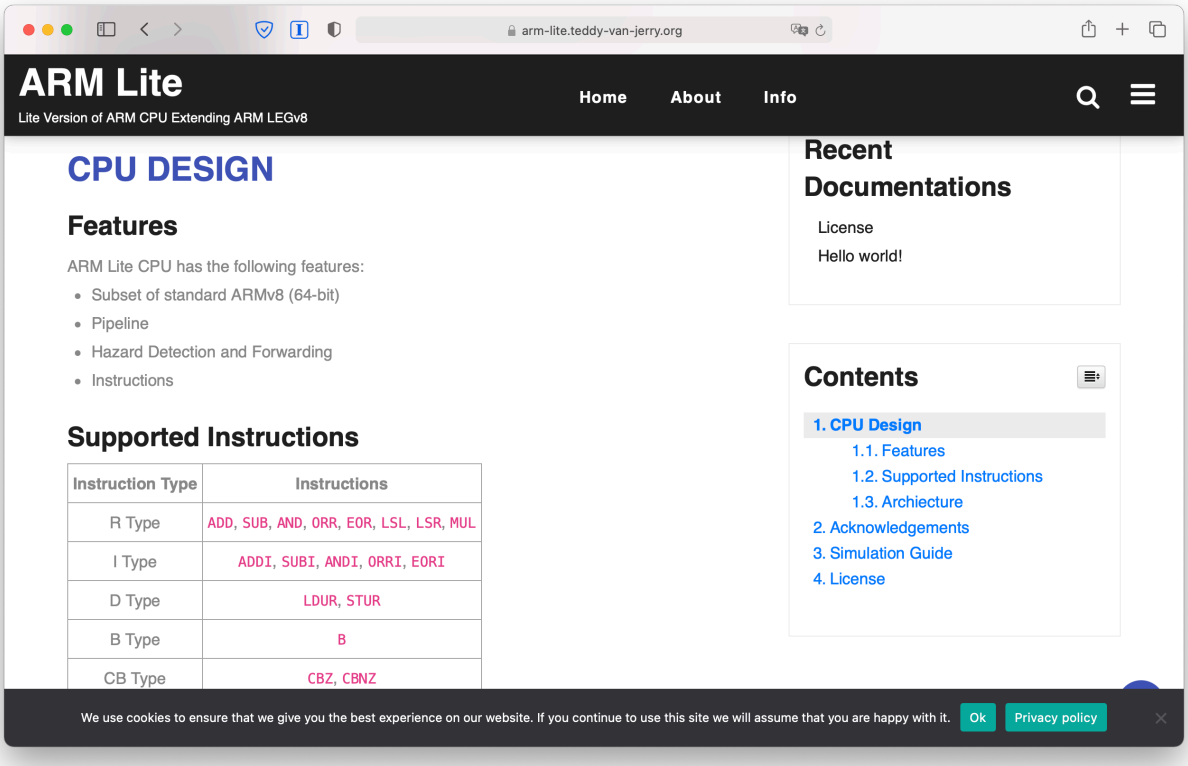



**Figure A.1:** ARM Lite website.